AFRL-SN-WP-TR-2002-1086

DEPENDENCY LANGUAGE REPRESENTATION USING CONCEPTUAL GRAPHS

Autonomic Information Systems

Harry S. Delugach
Lisa C. Cox
David J. Skipper

Bevilacqua Research Corporation
P.O. Box 14207
Huntsville, AL 35815

AUGUST 2001

Final Report for 19 September 2000 – 22 August 2001

20020307 047

SENSORS DIRECTORATE
AIR FORCE RESEARCH LABORATORY
AIR FORCE MATERIEL COMMAND
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7318

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE *(DD-MM-YYYY)* August 2001	**2. REPORT TYPE** Final	**3. DATES COVERED** *(From - To)* 19 Sep 2000 – 22 Aug 2001
4. TITLE AND SUBTITLE DEPENDENCY LANGUAGE REPRESENTATION USING CONCEPTUAL GRAPHS Autonomic Information System		**5a. CONTRACT NUMBER** F33615-00-C-1742
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER 69199F
6. AUTHOR(S) Harry S. Delugach, Lisa C. Cox, David J. Skipper		**5d. PROJECT NUMBER** ARPS
		5e. TASK NUMBER NZ
		5f. WORK UNIT NUMBER 06
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Bevilacqua Research Corporation, P.O. Box 14207, Huntsville, AL 35815		**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) SENSORS DIRECTORATE, AIR FORCE RESEARCH LABORATORY, AIR FORCE MATERIEL COMMAND, WRIGHT-PATTERSON AFB, OH 45433-7318		**10. SPONSOR/MONITOR'S ACRONYM(S)** AFRL/SNZW
		11. SPONSOR/MONITOR'S REPORT NUMBER(S) AFRL-SN-WP-TR-2002-1086

12. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This study is founded on the assumption that understanding the complex dependencies in large mission critical systems is a requirement for predictable, safe long-term operation of such systems. In the absence of a standardized language to represent dependencies, this study investigates Conceptual Graphs to determine if they are capable of representing dependencies and if they show traits that make them a suitable representation for dependencies. The fundamental issue was developing a definition of dependencies among system and software components, then finding suitable representations of the definition. This is a precursor to developing a formal language for dependencies.

15. SUBJECT TERMS:

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** Martin R. Stytz
a. REPORT Unclassified	**b. ABSTRACT** Unclassified	**c. THIS PAGE** Unclassified	SAR	82	**19b. TELEPHONE NUMBER** *(include area code)* 937-255-5900 x 3578

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. 239.18

DEPENDENCY LANGUAGE REPRESENTATION USING CONCEPTUAL GRAPHS

Harry S. Delugach
Computer Science Department
University of Alabama in Huntsville

Lisa C. Cox
Computer Science Department
University of Alabama in Huntsville

David J. Skipper
Bevilacqua Research Corporation

ABSTRACT / EXECUTIVE SUMMARY

This study is founded on the assumption that understanding the complex dependencies in large mission critical systems is a requirement for predictable, safe long-term operation of such systems. In the absence of a standardized language to represent dependencies, this study investigates Conceptual Graphs to determine if they are capable of representing dependencies and if they show traits that make them a suitable representation for dependencies. The fundamental issue was developing a definition of dependencies among system and software components, then finding suitable representations of the definition. This is a precursor to developing a formal language for dependencies. We started by postulating an English language definition for a dependency. We then examined, among others, Conceptual Graphs, Unified Modeling Language, Fault Trees, and various statistical approaches to capturing the meaning of dependencies. We conclude that Conceptual Graphs are capable and suitable for representing dependencies among system and software components.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

DEPENDENCY LANGUAGE REPRESENTATION USING CONCEPTUAL GRAPHS

Harry S. Delugach

Lisa C. Cox

David J. Skipper

Summary

The problem being addressed is the definition and representation of dependencies among system and software components. Dependency is any situation involving two or more elements where a change in one or more elements leads to a potential for change in one or more other elements. We have made a detailed study of the nature of dependency, from both a philosophical and a technical point of view, as a prelude to defining a formal language for dependencies. Due to our specific technical interests, a first concern was to determine if Conceptual Graphs were both capable and suitable method of defining and representing dependencies. This was accomplished by defining dependencies, then demonstrating that Conceptual Graphs could represent the definitions and could create examples of dependencies.

The importance of the work is potentially far-reaching. Larger and larger systems involve increasing numbers of components, so that the possible internal interactions become both more numerous and also more varied in their effects. Further, as systems become more and more interconnected, these system level connections become more numerous and varied. It is becoming apparent that testing, in the operational sense, cannot exhaustively test all interactions. Indeed, testing may never reveal all the dependencies, or even reveal all the important ones. Testing must be supported by some insights into the system structure and guidance as to the key dependencies. We believe the first step toward understanding dependencies is the ability to effectively represent them.

There are other approaches to dependencies that deal in what we call "shallow semantics." These approaches do not define or explore the meaning of dependency instead they simply state that it exists. Some approaches simply draw a directed arc, and call it dependency. Other approaches attempt to "quantify" the dependency without knowledge of the causality, either through Bayesian nets or probabilistic approaches. While these approaches contain numbers, which helps refine the notion of dependency, but it is not clear where such numbers have come from or what they really are measuring.

We start by grappling with the semantics of dependency and creating a definition and an ontology of dependency. Our representation of dependencies is a rooted approach, based on conceptual graphs, which is a general knowledge modeling approach. An advantage of conceptual graphs is that dependency information does not have to be "added on" syntactically or, more importantly, semantically to some existing representation, which may not be suited for that purpose. We also gain the further advantage that conceptual graphs are a formal logic and therefore are amenable to automated knowledge-based reasoning to support further analysis.

Our future plans are to further refine our dependency model in conceptual graphs and validate it with respect to large-scale models of software systems. After that we naturally expect to implement the language and develop support mechanisms for system developers and testers.

DEPENDENCY LANGUAGE REPRESENTATION USING CONCEPTUAL GRAPHS

Harry S. Delugach

Lisa C. Cox

David J. Skipper

1. INTRODUCTION AND OVERVIEW

This section introduces the research topic and provides a brief overview of the work on this project. In summary, this research approaches language definition by starting with a definition and a representation of a definition of dependencies. In the course of the research, several potential representation methods were identified, but an approach based on Conceptual Graphs (CGs) was selected as the basis of a representation based on the richness of the representational capability and the formal basis available to a language.

1.1 Problem Discussion

Today, systems engineers are creating both larger and smaller systems with an increasing number of components, so that the possible internal interactions become both more numerous and also more varied in their effects. As systems of systems arise, more and more interconnections exist between the systems in these collections, and the system's level connections become more numerous and varied. Random exhaustive testing becomes too expensive, thus forcing a more focused testing with greater front-end analysis support. Indeed, it becomes less likely based on sheer numbers, that testing will reveal all the dependencies and; in fact, testing may not even reveal all the important ones. Testing must be supported by some insights into the system and guidance as to the key dependencies and their characteristics. We believe the first steps toward understanding dependencies in systems are to define dependencies, then to effectively represent them. Given a representation or even a formal language, dependencies could be described and engineered as the system is designed. Ideally then automated tools would analyze the stated dependencies to determine their interactions and to discover, or mine, new dependencies. The implication here is that the representation or language should have some formal logical basis rather than just a basic symbology.

Thus the problem being addressed is a representation of dependencies within and among system and software components in a form that is useful for formal manipulation by either humans or computers. Use by computers assumes that there exists a clear, unambiguous formal language for dependencies that is suitable for computer usage. Lacking such a language, one could be created, or appropriated from some other field, as long as the language met the stated requirements for a dependency representation. Ideally, some industrial or government group would provide an approved requirements document for this language as has been done in the past developments, *e.g.* the Ada language. From such a document, determination of the suitability of an existing representation (such as Conceptual Graphs) would be straightforward. Unfortunately, there was not such a document provided for this study, nor does one appear to exist. However, this study will not develop a dependency language or even a requirements document for such a language. Rather, this study will investigate some preliminary, basic issues that must be considered before such work can proceed and at the same time investigate the suitability of a specific representation, Conceptual Graphs.

The first stumbling block seems to be that there is no exhaustive definition of the meaning of dependencies, especially in a manner amenable to formal language definition. Consequently, the first issue dealt with in this study was a substantive definition of dependency (mostly a semantic issue). This issue is followed by additional issues that include:

What formal representation should be used for the definitions? (Mostly a syntax issue)

What requirements are implied by the definitions? The intended usage?

What additional language requirements are implied by the previous requirements?

What subset of the language can be implemented on a computer?

What about verification, validation, provability and testability?

These issues would have been resolved before a formal language definition requirements document was created. Such a document clearly should form the basis a dependency representation. In the absence of such a document, this study is constrained to the essentials, or first a finding definition of dependencies. This will provide a basis for a formal requirements definition and language development in the future and a basis for examining existing representations.

1.2 Solution Approach

Faced with this formidable array of issues, a limited time frame and a limited budget, we selected a straightforward, established approach to this problem. First, focus on the true basics, *i.e.* crafting a workable definition of dependencies, followed by a representation of the definition in a more tractable form, which will pave the way for detailed requirements definition. We proceeded by postulating a basic meaning for a dependency in English, then extending the meaning with further definitions, and eventually using a well-established knowledge modeling technology to clarify the definitions. We then examine other representations that appear to describe dependencies for a comparison to our definition and representation. We finally examine some real world problems as a demonstration that the technology involved was suitable and capable of describing dependencies .

Starting in the dictionary [Web80]:

"dependency ... 1: DEPENDENCE 1 ..."

"dependence ... 1: the quality or state of being dependent; *esp*: the quality or state of being influenced or subject to another ..."

"dependent ...2 a: determined or conditioned by another : CONTINGENT b: relying on another for support c: subject to another's jurisdiction ..."

Starting from this point, a rough, imprecise English language description of a dependency is constructed from this, influenced by the systems view that only the attributes of objects are visible:

A dependency is a situation involving two or more elements where a change of state in one or more elements leads to a potential for a change of state in one or more other elements.

This definition formed the starting point for this study, with "situation", "elements", and "potential" being key components. By extending and examining the examples, Conceptual Graphs are shown to be capable of representing dependencies. Further, based on criteria developed from previous research, they appear to be a suitable method to represent dependencies.

2. BACKGROUND

2.1 Introduction

This research builds upon lessons learned and directions taken in other research efforts. The following section describes the key efforts in past work in defining and representing dependencies along with identifying weaknesses and strengths of each. While research specifically aimed at the definition of dependencies was not uncovered, several research elements contained definitions or representations of dependencies. These are examined to provide insights into the problem.

2.2 Previous Research Identified

This section identifies previous work in modeling dependencies and summarizes certain key approaches.

2.2.1 Unified Modeling Language

In modeling, it is usually important to identify, characterize, and understand the impact of the dependencies that exist between the entities in the model. This is vital at all levels of modeling and in all domains. The concepts and approach presented in this report are applicable at all levels and in each domain. We start by considering that there are many notations in use for the specification of and analysis of models. While some of these notations allow explicit specification of dependencies, most appear to include dependency only by implication. For example, the Unified Modeling Language (UML) defines a basic dependency and then represents a simple dependency as a dotted arrow between components as shown in Figure 1. [Boo99] Also refer to section 4.0 for further details on dependency definitions in UML.

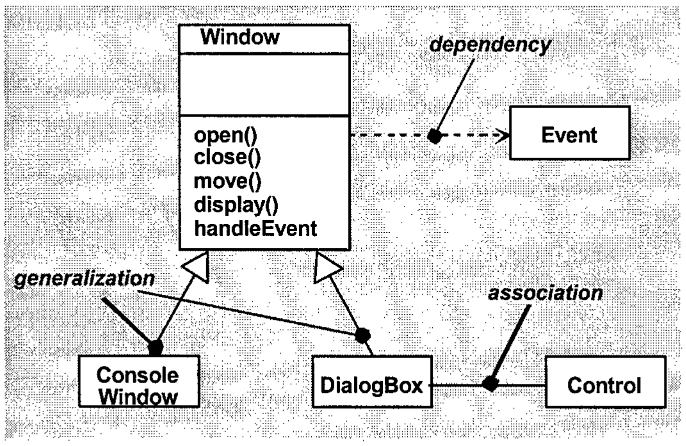


Figure 1. UML relationships.

2.2.2 Cyc

The Cyc project [Len95] has made a substantial impact on the modeling community. Cyc is a large collection of defined types and relationships, supported by a rule-based expert system. Although it is the result of years of effort, its main focus is on having broad coverage; hence its treatment of dependencies (and relationships in general) is somewhat shallow.

2.2.3 Other Approaches

In the software engineering domain, Microsoft (MS) is presently advocating the Open Software Description (OSD) as the standard for describing and packaging software [Hal97]. While OSD and its surrounding literature are in agreement that dependency representation is important, OSD utilizes a simple viewpoint and simply represents dependencies by supplying a list of other components that are required to be present before a particular software component can be installed on a system [Han99].

Work has also been done in the natural language processing area dealing with dependency analysis between words of a sentence, and specific linguistic dependency types have been identified, such as the dependency between a noun and a determiner or the dependency between noun and verb [Sle91], [Che97]. However, these dependencies once again are limited to the particular domain in question (*i.e.*, linguistic dependency) and explicit definitions of an abstract dependency are not considered.

Much of the present literature takes the definition of dependency for granted and where definitions are occasionally given, they vary widely. Some sources maintain that dependencies are simply first-order logic formulae, or in database terminology, constraints [Tha98], [Cre00], [Hal97]. Others claim that higher-order logic is required to express dependencies. [Pro00] Some take a probabilistic approach and express dependencies as conditional probabilities between specified variables or look solely at dependencies from a statistical viewpoint. [Lev00], [Sub00], [Bri98] Some sources take the approach that a dependency is best modeled by the client/server relationship, and then develop the definition of dependency in client/server terms [Rum98], [Yu96], [Kel00], while others specify types of dependency such as structural and functional dependencies [Kel00] or data and value dependencies. [Pap97]

Keller, Blumenthal, and Kar in [Kel00] attempt a more in-depth characterization of dependencies and define six different "dimensions" of dependency, and Prost in [Pro00] also takes a "type-based" approach to dependency analysis. However, some of the dimensions given in [Kel00] for analyzing dependencies are actually attributes of the computer system under analysis, not relationships between its components. Once again, there is no clear delineation between the dependency itself, and the domain in which the dependency exists.

Mineau in [Min94] discusses the addition of functions to and the treatment of functional dependencies in Conceptual Graphs, but even Mineau does not address the explicit definition of a dependency.

This report presents an approach for formally defining and characterizing dependencies using Conceptual Graphs. It is our contention that our approach to the definition of dependency and the use of Conceptual Graphs as a dependency language allows for a much more coherent and complete description of dependencies at the general level and explicitly delineates the characteristics of the dependency from any domain limitations. We also expect the use of Conceptual Graphs to allow more powerful analysis of the dependencies of a given system.

2.3 Significant Past Work Summary

Here we present several possible existing solutions and describe why we believe they are insufficient for the practical handling of large-scale and richly described dependencies. We briefly describe four of these possible solutions. We adopt the position that the better solutions will involve representations that have the potential for comprehensively specifying a wide range of computer systems' characteristics, rather than simply focusing on their dependency characteristics. We could choose to provide a language solely for describing dependencies and still fulfill our obligations under this contract; however, such a language would require additional resources and infrastructure in order to be useful in the wider area of computer systems development and deployment. Both UML and Cyc have the potential to describe many characteristics beyond mere dependency. We focus our attention on these two, and spend less time discussing fault tree analysis and statistical techniques, not because the latter are not useful, but because we believe dependency analysis should be integrated into the overall process of system specification and development.

For each of the approaches, we present a brief summary describing the approach, its main use in dependency analysis, its main shortfalls with respect to dependency analysis. See section 4 for a more detailed description and evaluation.

2.3.1 UML Dependencies

Dependencies are represented in a variety of ways in UML, Figure 1 and Figure 2, most of them implicit. UML Associations are dependencies. UML Constraints are a type of dependency. UML References, Aggregations, Figure 2, and Generalizations, Figure 1, are types of dependencies. Because the dependency relationship is so complex, it is listed among what are termed "advanced relationships" in UML [Boo99]. The practice of labeling associations and dependencies is an attempt to include ad-hoc semantic information. This is an example of the difficulty faced by the modeler when trying to distinguish and classify the dependencies.

The UML "dependency" construct is a "using" relationship stating directional links between classes. When one class uses another, a change in the latter may require a change to the former. This is a true dependency, however the specifics relating to the dependency are still imprecise, hence the need to label the dependency arrows in a UML model with additional information concerning the dependency type.

The UML "Association" is the definition of a peer-to-peer relationship between classes. The Association is often used to represent the semantics of a dependency. The names, roles, multiplicity, and aggregation (called "adornments") add further information about the dependency's semantics; however, the specifics of the dependency itself, such as the "need" implied by the dependency are still imprecise.

"Generalization" refers to the links between generalized classes and specializations of those classes. This corresponds to what we have named "Inheritance Dependency". The specialization, or the inheriting class is dependent upon the definition of the generalized class. This is an implied structural dependency whose semantics are generally implementation-based in practice; that is, inheritance is used as a form of structural re-use and therefore shows how one class depends on another's structure.

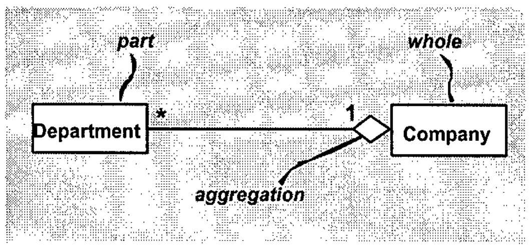


Figure 2. UML Aggregation relationship.

The UML "Aggregation" involves a PartOf dependency. There is also a stricter form of aggregation, called "composition" that denotes when a part "lives and dies" with the whole. For example, simple aggregation a person is part of a company, but has an existence outside the company's existence; whereas a person's arm is part of that person and cannot reasonably be thought to exist separately.

UML has extensibility mechanisms to extend its notation; however, they do not in general extend any semantics. These mechanisms are stereotypes and constraints. Stereotypes permit relationships (as well as other constructs) to be labeled as to their "type". In UML, for example, "access" or "reference" stereotypes imply a functional dependency. For the most part, a stereotype consists of either a simple named type of relationship, or else it corresponds to some definition that really is a template for that relationship. No new semantics are introduced.

A UML constraint is also called an "extensibility mechanism" in that it extends the standard UML representation to allow specification of additional semantics of the model. It is a dependency at a lower level of detail. The example given in [Boo99] concerning the required order of addition is expressing a dependency between the addition function and the order of the data presented. This expresses what we call a Constraint Dependency between function and data. UML constraints are expressed in the OCL constraint language that forms annotations somewhat "outside" the UML representation itself. That is, they require separate processing and interpretation. There are several other dependencies implicit in various other UML diagrams. Refer to section 4 for further discussion.

2.3.2 Cyc's Dependency Representation

The ontology of Cyc [Len95] has made a large impact in the modeling community. As a large-scale, industrial-strength effort, it has the potential to cover a substantial part of human knowledge. We have studied Cyc's notion of dependency in detail, Figure 3. The main points we have found are:

- Cyc does not differentiate between relations and dependencies, thus leaving any dependency knowledge implicit
- Relations between entities are expressed by defining a RelationType, a Relationship, or Predicate.

This approach is typical of the approaches we have found. Each relation has its own semantics, which may involve varying degrees and kinds of dependencies. In order to perform dependency analysis, each one of the relationship categories must be analyzed to determine exactly what the relationship means and what its dependency semantics are. It is also likely that not all modelers will use the same relationship in exactly the same context, implying that the actual dependency semantics may be different for the same category used by different modelers.

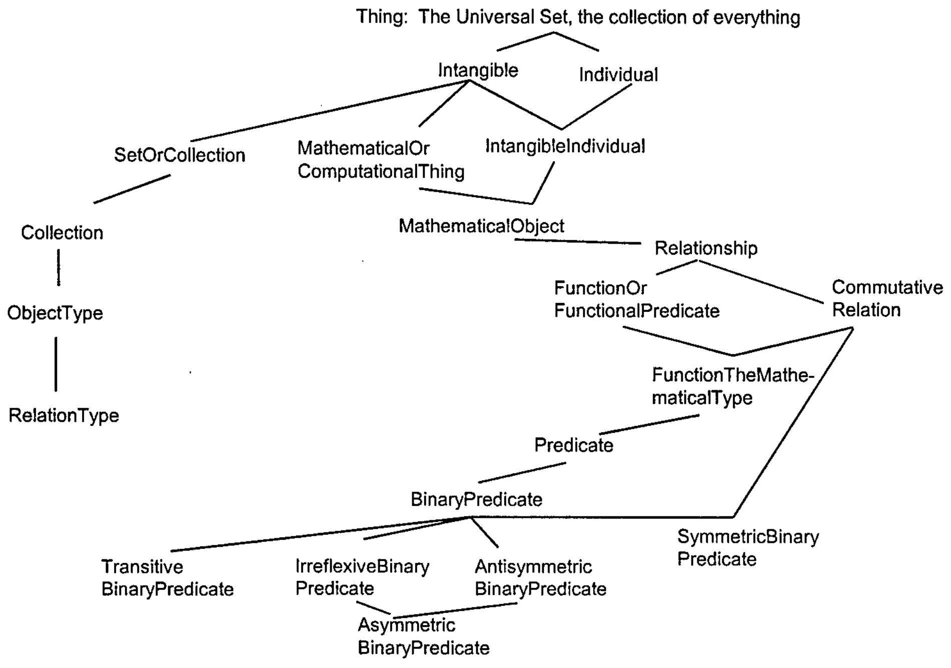

Thing: The Universal Set, the collection of everything
Intangible
Individual
SetOrCollection
MathematicalOr ComputationalThing
IntangibleIndividual
Collection
MathematicalObject
Relationship
ObjectType
FunctionOr FunctionalPredicate
Commutative Relation
RelationType
FunctionTheMathematicalType
Predicate
BinaryPredicate
SymmetricBinary Predicate
Transitive BinaryPredicate
IrreflexiveBinary Predicate
Antisymmetric BinaryPredicate
Asymmetric BinaryPredicate

Figure 3. Cyc's Upper Level Ontology With Respect To Relationships.

The partial hierarchy shown in Figure 3is that portion of Cyc's ontology dealing with relationships. Note that there are a number of relationship names, but little distinction between relationships. Furthermore, relationships are dealt with more mathematically and hence are less able to be used in modeling general dependencies within relationships.

2.3.2 Fault tree analysis

Fault tree analysis is a well-known operations research technique to analyze component systems. It allows successive failures in a system to be analyzed, so that component points with a high resulting risk of failure can be pinpointed. The difficulties of using fault-tree analysis are also well known, particularly when it comes to large-scale systems.

2.3.3 Statistical Models

Statistical models (*e.g.*, Bayesian networks) are used in an attempt to capture more interactions between components than fault trees. In a statistical model, the probability of failure of a component in linked to other components via probabilistic factors, so that a particular component's failure probability is a function of other components' failure probabilities. The drawbacks of such an approach are:

Fundamental – Bayesian methods require mutually exclusive and exhaustive hypotheses to meet the underlying mathematical requirements. Approximations, or even ignoring the requirement, may lead to non-intuitive or even incorrect results.

Semantic – Bayesian methods are measures applied to sets of events. There is no explicit representation of what is being measured or why it is the value selected. Further, because probabilities in Bayesian representations are point values, there is no explicit representation of ignorance. Clumsy add-ons must be inserted to cope, but again there is no explicit relationship to properties.

3. METHODS, ASSUMPTIONS, AND PROCEDURES

3.1 Capability Overview

As discussed in the introductory material, the methodology is driven by the paucity of formal requirements. Thus determination of the capability and suitability of Conceptual Graphs for dependency representation is forced to adopt a familiar approach, postulate a basis, develop related definitions and then demonstrate satisfaction of the definitions. The first of these, capability, was accomplished in the introduction. Using this as a start, we have made a study of the nature of a dependency, from both a philosophical and a technical point of view. We then examined representations of this fundamental definition to determine if they insights or advantages for machine manipulation. However, we did eschew a purely formalized mathematical approach as might be found in formal systems theory. The rationale for not pursuing that alternative was that the major emphasis in our work was to establish the ability to use dependency modeling based on Conceptual Graphs to support a full-blown model of a software system for computerized manipulation. This should not rule out a future mathematical study of the subject to establish a rigorous foundation for future requirements standardization. We therefore explain in following sections how a conceptual graph system works and how dependencies are naturally defined in such a system formal logic system. As an exercise, we generated example problems to demonstrate the definition and the representation approach. Interestingly, because we chose to utilize a formal logic knowledge representation that is capable of machine manipulation, it should, in future work, be possible to compile a compendium of definitions and related definitions for machine analysis to "mine" or discover underlying or hidden knowledge about the meaning of dependencies to further support standards development.

3.2 Suitability Overview

The previous section provided an overview of the methodology to determine the capability of Conceptual Graphs to represent dependencies. This leaves the issue of suitability of Conceptual Graphs for representing dependencies. In the absence of other techniques in a given domain, capability implies suitability. In the presence of other techniques in a given domain, a comparison is performed to prove or at least to demonstrate that Conceptual Graphs perform at least as well as the other techniques on the average in that domain. The statement of work for this study does not require proving or demonstrating superiority of the method of Conceptual Graphs with respect to other methods. The following sections develop a set of representation characteristics, based on the previous definition and the authors' perceptions of domain independence to as great a degree as possible, needed to provide a suitable representation to support this effort.

3.3 Overview of Desired Characteristics of Dependency Representation

From the previous study, certain attributes of a dependency representation appear to be desirable. This is a list of the most pertinent ones identified.

Traceable to a standard definition, for both the meaning and the supporting symbology

Explicit Syntax and Semantics features, to include:

> Quantification, with or without Uncertainty, of amounts present or required for the dependency
>
> Causality, both singleton events and aggregated events with or without temporal or spatial synchronization
>
> Transitivity to represent the flow and allocation of amounts of dependency in a system
>
> Composition and Simplification, as well as Generalization and Specialization to provide extensibility
>
> Procedures and Sequences
>
> Time to include elapsed, relative, and absolute
>
> Abstractions and Instances of dependencies (dual hierarchy language)
>
> Embeddable in situational and contextual representation to portray the limitations and validity of the dependency

Clear logic operations to manipulate the representation of dependencies and query representations.

There are also general properties that are less easy to quantify. These are however, still important to a language designer.

Compact efficient representation

Flexible, but unambiguous representation

Standardized representation

Costs, as low as possible for adoption, training and maintenance.

4. ALTERNATIVE SOLUTIONS AND EVALUATION

4.1 Summary of Alternatives

The brief summary of current approaches makes it clear that fault-tree analysis and statistical methods seem insufficient for the task at hand. Their lack of explanation capabilities, and difficulties in modification are important limitations. While there is surely a set of limited applications where they can be effectively used, they appear inadequate for the general domain of large, complex, and dynamically changing systems. We therefore focus our discussion on UML and Cyc, which appear to be more attractive candidates for representing dependencies. We begin by extending our discussion of UML, then follow by providing an overview of the strengths and weaknesses of these approaches.

4.2 UML Dependency Alternative

Since the Unified Modeling Language (UML) is so widely used, a major focus of our effort has been to determine whether UML is sufficient for representing the dependencies required. It is our opinion that UML itself is not adequate for the task. It is important for any modeling approach to compare itself with UML, for several reasons. First, UML is very popular in systems and software modeling endeavors. It is clearly perceived as useful and effective in many environments. Secondly, because of its popularity, there are many UML legacy models already in use; it behooves any modeler to have the ability to leverage the use of all those models. Finally UML can form a common "language" for describing models, thus facilitating communication among not only modelers themselves, but also other stakeholders in a system. We will now explain why we have determined that UML is not sufficient for the task, although we intend that our approach be able to support translation to and from UML descriptions to leverage UML's current popularity among developers.

Dependencies in UML are spread out over all of the UML diagrams, and generally implicit. Object and class diagrams contain several different kinds of dependency, each with its own characteristics (e.g., presence or absence of transitivity) and set of rules. This section outlines all of the models in UML and briefly describes their dependency representations.

While it may be useful to pursue a detailed analysis of each kind of dependency in each kind of diagram, our point here is that there is no uniform representation for dependency in UML, nor is there a way to specify dependency semantics beyond UML's original semantics. If our survey of UML is not exhaustive, we wish to make the point that additional features and subtleties of UML merely make a unified dependency approach harder as more features must be somehow incorporated in an ad-hoc way.

4.2.1 Class and Object diagrams

Dependencies are represented in a variety of ways in UML, Figure 4, most of them implicit. UML Associations are dependencies. UML Constraints are a type of dependency. UML References, Aggregations, Figure 5, and Generalizations, Figure 4, are types of dependencies. Because the dependency relationship is so complex, it is listed among what are termed "advanced relationships" in UML [Boo99]. The practice of labeling associations and dependencies is an attempt to include ad-hoc semantic information. This is an example of the difficulty faced by the modeler when trying to distinguish and classify the dependencies.

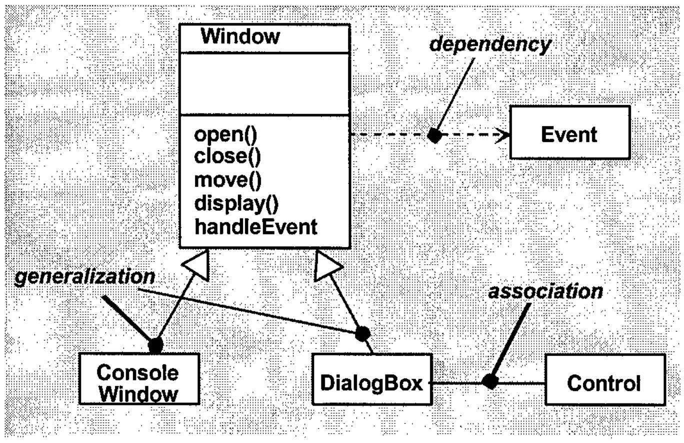


Figure 4. UML relationships.

The UML "dependency" construct is a "using" relationship stating directional links between classes. When one class uses another, a change in the latter may require a change to the former. This is a true dependency, however the specifics relating to the dependency are still imprecise, hence the need to label the dependency arrows in a UML model with additional information concerning the dependency type.

The UML "Association" is the definition of a peer-to-peer relationship between classes. The Association is often used to represent the semantics of a dependency. The names, roles, multiplicity, and aggregation (called "adornments") add further information about the dependency's semantics; however, the specifics of the dependency itself, such as the "need" implied by the dependency are still imprecise.

"Generalization" refers to the links between generalized classes and specializations of those classes. This corresponds to what we have named "Inheritance Dependency". The specialization, or the inheriting class is dependent upon the definition of the generalized class. This is an implied structural dependency whose semantics are generally implementation-based in practice; that is, inheritance is used as a form of structural re-use and therefore shows how one class depends on another's structure.

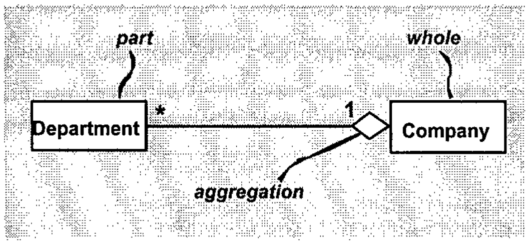


Figure 5. UML Aggregation relationship.

The UML "Aggregation" involves a PartOf dependency. There is also a stricter form of aggregation, called "composition" that denotes when a part "lives and dies" with the whole. For example, simple aggregation a person is part of a company, but has an existence outside the company's existence; whereas a person's arm is part of that person and cannot reasonably be thought to exist separately.

4.2.2 Sequence and Collaboration Diagrams

Interaction diagrams in UML are of two types: sequence or collaboration diagrams. Their properties are similar, so we will discuss sequence diagrams here. Figure 6 is an example of a sequence diagram. The presence of a message passing between two objects is shown by a labeled arrow. There are several kinds of dependency shown here, each with its own semantics and representation.

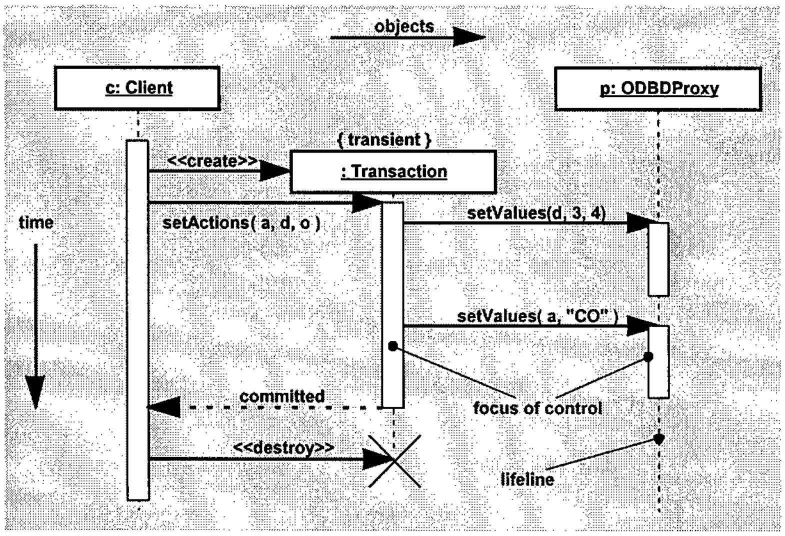


Figure 6. UML Sequence Diagram.

There is an initiation dependency: the message **<<create>>** causes an instantiation of the class **Transaction**. Note that it is not the **<<create>>** message itself that conveys initiation dependency semantics; it is the appearance of the **Transaction** object as result of the **<<create>>** message.

Access dependencies are shown by several messages; e.g., the **setValue** messages show that the **Transaction** object is dependent on the **OBDProxy** object.

A temporal dependency is shown by the **Client** object receiving the **committed** signal at the end of the **Transaction**'s processing. Since UML's semantics for a message is to transfer the focus of control, we assume the **Client** waits for the **Transaction** to finish.

The parameters to a message also represent an access dependency on the structure of the parameter class. For example, the parameters to **setActions** are each of different types (not shown here) whose structural descriptions create a dependency of both **Client** and **Transaction** upon their type definitions.

There are other dependencies inherent in such a diagram. Our point here is that each of them must be modeled in a separate way and their semantics captured in an ad-hoc fashion. There is no unified semantic model for all these dependencies.

4.2.3 Use Case Diagrams

UML's use case diagrams are often used at the beginning or top-level of a specification process, since they are a flexible and understandable means of representing goals in a system. For dependency analysis, however, they are rather limited in their expressive capabilities. Some relationships between use cases strike a parallel with relationships between classes and instances; e.g., generalization suffers the same limitations with respect to dependency in use case diagrams as it does for class and object generalization. Consider some examples as in Figure 7.

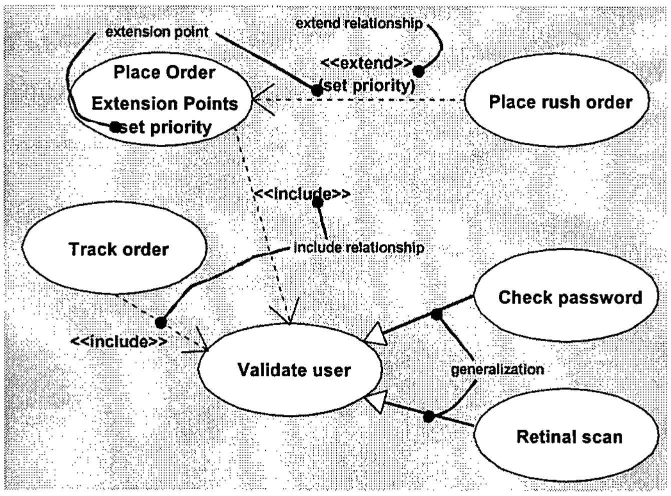


Figure 7. UML Use Case Diagram.

The **<<include>>** stereotype constitutes a case of procedural aggregation, as in a part-whole relationship. It thus constitutes both a structural dependency and a need dependency. For example, the **Track order** use case needs the **Validate user** use case for its operation.

The **<<extends>>** stereotype relationship provides a more interesting form of dependency; namely, periodic or occasional dependency, since the extension case does not always occur. Once again, this is a different kind of relationship from any previous ones, and its semantics must be established from scratch.

An extension point in a use case provides a clue that there will be a dependency, but like a routine invoked in a programming language, it does not specify the component(s) on which the use case depends. For example, specifying **Place Order** with an extension point does not show on what use cases the **Place Order** use case depends.

Another important limitation to dependency analysis from use case diagrams is not apparent here: the lack of a mapping from use cases to deployed software components. Of course, many UML modeling environments provide such a mapping, but it is the software development environment, not the model itself, which permits a correspondence between use cases and the collaborations (collections of classes and instances) which implement the use case.

Another limitation is the lack of semantics for the individual use cases, since they are denoted by phrases that can allow several different meanings.

4.2.4 State Diagrams

State diagrams in UML are adapted from Harel's statecharts. Again, they contain implicit dependencies, each of which must be separately analyzed and modeled. Consider the example diagram in Figure 8.

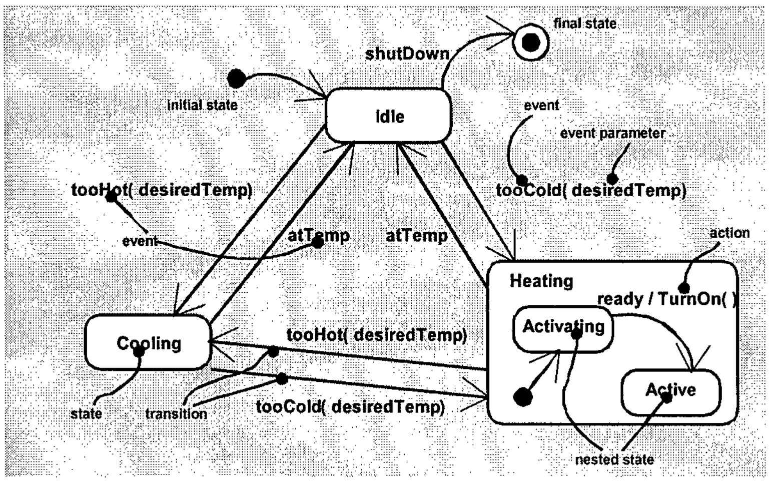


Figure 8. UML State Diagram.

As could be expected, state diagrams contain many temporal dependencies, in particular imposing constraints on the order in which operations and events may meaningfully occur. This kind of temporal dependency has to be separately modeled from other temporal dependency types in other diagrams (e.g., sequence diagrams) and therefore requires additional ad-hoc analysis. For example, the **TurnOn()** operation must be preceded by a **tooHot(desiredTemp)** event, whether in the **Idle** state or the **Cooling** state. There is also little semantic information that can be used to relate the dependencies in this diagram to dependencies in other diagrams.

4.2.5 UML Results

UML has extensibility mechanisms to extend its notation; however, they do not in general extend any semantics. These mechanisms are stereotypes and constraints. Stereotypes permit relationships (as well as other constructs) to be labeled as to their "type". In UML, for example, "access" or "reference" stereotypes imply a functional dependency. For the most part, a stereotype consists of either a simple named type of relationship, or else it corresponds to some definition that really is a template for that relationship. No new semantics are introduced.

A UML constraint is also called an "extensibility mechanism" in that it extends the standard UML representation to allow specification of additional semantics of the model. It is a dependency at a lower level of detail. The example given in [Boo99] concerning the required order of addition is expressing a dependency between the addition function and the order of the data presented. This expresses what we call a Constraint Dependency between function and data. UML constraints are expressed in the OCL constraint language that forms annotations somewhat "outside" the UML representation itself. That is, they require separate processing and interpretation.

Here is the summary of UML's characteristics with respect to dependency.

UML Dependency Representation	
Use(s)	Specifying software and computer systems
Strength(s)	Generality; diagrams cover a wide range of system characteristics
Weakness(es)	**No underlying built-in semantics for dependencies.** **Several different ways to represent dependencies.** **Lack of a formal semantic description for UML in general**
Potential	Modifications to UML are possible but provide no additional semantics without programming.

Table 1 UML Dependency Representation Capability

No underlying built-in semantics for dependencies. The dependency arrow itself is the best illustration of this. It may be considered to represent a general "functional dependency" but it isn't clear what a user of a UML diagram is supposed to do with it – what constraints must therefore be followed, or what additional meaning is implied by the dependency. For simple traceability analysis, dependency arrows are helpful in identifying where changes may be implied, but there is no further guidance to a UML modeler as to what kind of changes the dependency implies, etc.

Several different ways to represent dependencies. Because there are a number of UML relationships that represent implicit dependencies, the notion of dependency is distributed among a number of different language constructs. This is not just a matter of inconvenience; it also means that there exists, for each construct, a variety of differing interpretations, each one of which affects what the dependency may mean. This makes sharing knowledge about the dependencies all the more problematic.

Lack of a formal semantic description for UML in general. Even if the problem of ill-defined dependency semantics could be solved for UML, we still have to deal with the fact such semantics are embedded within the larger UML semantics, which are ill defined for the purposes of formal analysis. Therefore the only way to achieve a well-defined formal semantics for dependencies in UML would be to establish a well-defined formal semantics for the entire UML representation! Such a task would be a large and years-long undertaking.

Of course, modifications and extensions to UML are always possible; however, these extensions (while expressive) provide no additional semantics to a UML description without changes to the UML processing system underneath. That is, any tools that want to use an extended UML description will have to be modified internally in an ad-hoc way to support those new semantics. Using our approach, these dependencies are expressly described and the semantics are formally included in the ontology so that they can be used in modeling the software requirements.

4.3 Cyc Dependency Alternative

Here is the summary of Cyc's appropriateness to dependency.

Cyc Dependency Representation	
Use(s)	Modeling "common-sense" knowledge, modeling of computer systems
Strength(s)	Extremely large-scale, covering large portions of many aspects of human knowledge
Weakness(es)	Limited semantics Many ad-hoc distinctions for organizational purposes Lack of ability to describe processes
Potential	Can address many subtle points in dependency, for example in human computer interaction, that require deep knowledge

Table 2 Cyc Dependency Capability

Limited semantics. Cyc's semantics are limited to constraints within frames: attributes, some relationships, etc. There are no formation rules or chaining rules to transform representations.

Many ad-hoc distinctions for organizational purposes. Since Cyc has developed an extensive type hierarchy, many type names are not "natural types" in the sense that they are for convenience in inheritance or aggregation.

Lack of ability to describe processes. Since much of software specification involves describing processes, this is a significant shortcoming. Many dependencies are temporal or process-based and would therefore not appear in a strictly static description.

4.4 Fault-Tree Alternative

Here is the summary of fault-trees with respect to dependency.

Fault-Tree Dependency Representation	
Use(s)	Models component failures where failure rate can be assigned a number
Strength(s)	Allows successive failures to be analyzed
Weakness(es)	Limited semantics Requires pre-analysis of failure modes Lack of modifiability Composition is limited to transitivity

Potential	Scalability can be addressed with some optimizations Some partitioning may reduce modifiability problems

Table 3 Fault Tree Dependency Representation Capability

Limited semantics – the fault tree "knows" nothing about the systems it is modeling. If a component is very important or if it is minor, the model makes no distinction. There is little potential for reasoning about anything other than the cascade of fault modes.

Requires pre-analysis of failure modes – an analyst must carefully review all of the components

Lack of modifiability – related to the pre-analysis problem, once the tree is constructed it becomes very difficult to determine where new components should be added or what existing components' placement must be modified. Given the domain of large, reconfigurable systems, this is a serious inefficiency and source of errors

Composition limited to transitivity – while failures or compromises may have a rich set of interactions, in fault tree approaches there is only one way to combine them: composition. We believe the full story of dependencies is much more interesting than that, as we show in our approach using conceptual graphs and a variety of attributes for dependencies. See section on Transitivity Issues.

4.5 Statistical Models Alternative

Here is the summary of statistical models with respect to dependency.

Statistical Models Dependency Representation	
Use(s)	Component failures where enough observations allow probabilities to be assigned
Strength(s)	Efficient Well-understood Bayesian network research underlying the model
Weakness(es)	Lack of explicit causality knowledge Lack of detail Lack of explanation of results
Potential	As experience is gained, may be useful in already-deployed computer systems

Table 4 Statistical Model Dependency Representation

Lack of explicit causality knowledge – Failures of components are assumed to occur together (with a certain probability), whether or not there is any causal link between them. This makes it difficult to modify as well, because new components' failure rates may not be known, nor will their impact on other failure rates be derivable from the statistical model.

Lack of detail or explanation of results – Statistical models may generally model only the notion of "failure" as a Boolean condition; they cannot model different dimensions of failure (*e.g.*, graceful degradation) or degrees of failure. Furthermore, they lack any way to describe "why" a failure may have occurred; this does not permit a conceptual model of failure to be understood.

Finally, see Section 6 for a comparison to the proposed solution.

5. PROPOSED SOLUTION

5.1 Introduction to Conceptual Graph Technology

We will use conceptual graphs [Sow84] as our basis for a dependency language. Conceptual graphs are a formal, logic-based, semantic network language introduced in 1984 and further refined through numerous workshops and conferences (*e.g.*, [Luk97], [Che98], [Del01]). For the purposes of this work, we consider it a mature technology, well established in the modeling community. An interchange standard, the Conceptual Graph Interchange Format (CGIF) is being formulated and should be in place by the end of 2001. This will allow a conceptual graph system to exchange knowledge with other systems and thus leverage already-existing knowledge in performing dependency analysis.

This research effort was specifically aimed at developing the dependency language itself, without determining how dependency information would be acquired. We expect that acquiring the dependencies can be performed in several ways: from direct interaction with users who will write conceptual graphs, from translations to and from UML specifications and conceptual graphs, or from interchange with other intelligent systems that adhere to the CGIF format.

Our approach employs rich ontology of conceptual dependency, which we consider preliminary, but a good indication of what we believe the shape of an eventual ontology will be. One key consideration is that our representation will be extensible. In the structure of conceptual graphs, so that new dependency dimensions can be easily incorporated and handled by an analysis system.

5.2 Basic Terms

Our perspective comes from the Realist's view as defined by Hayes in [Hay94]. We assume "a set can be a set of anything" and that "the universe can be physical or abstract or any mixture" in order to make our universe as general as possible. [Hay94] Based upon this perspective, we then refer to an **entity** as anything that can be a member of such a set, and therefore can be anything we want to model. This can be an object, a concept, an organization, or any other thing to be modeled. We also make the assumption that the entities are not static. The entities can **change**. At this point, we simply assume the existence of something called change that happens to entities, but we deliberately do not yet attempt to define change in order that it, too, may be allowed to be as general as possible. We understand that an entity may change for at least several and possibly many reasons. The entity may have change as part of its very nature (for example, try to model a 2-year-old child without allowing for change). The entity may also be influenced to change by something outside itself. This type of change is of specific interest to us and it forms a basis for our understanding of dependency. From this understanding, we assume that there are cases where the "something outside itself" possibly or **potentially** influences the entity to **change**.

We propose our general definitions for **entity**, **change**, and **potential for change** as unproven axioms so that we may concentrate upon dependency. We also assume the existence of a relation R between some number of entities, shown by R(A, B, C, D, . . .) where it can be said (as a primitive statement) that the R relationship exists between the entities A, B, C, D, *etc.*

In the general case, we define a **dependency** as such a relation, D, between some number of entities wherein a **change to one of the entities implies a potential change to the others**. We can therefore show such a general dependency as D(A, B, C, D, . . .) where $D \in R$. This general form of a dependency is shown in Figure 9. In order to emphasize the complexity of this most general type of dependency (which may exist between many entities), we refer to it as symbiosis.

As an example of this most general type of dependency, or symbiosis, we can consider the relationship between the departments within a corporation. It is easy to see that the engineering, accounting, contracts, marketing, and facilities departments are dependent upon each other. However, it is not at all easy to specify and quantify the extent of such a dependency.

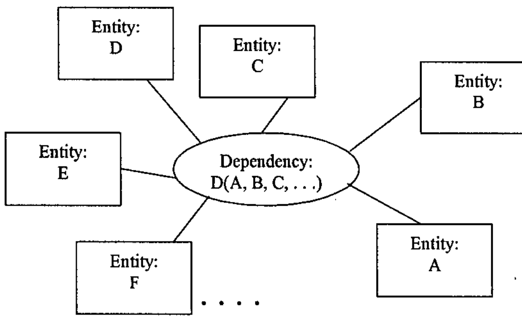


Figure 9. Graphical representation of most general form of a dependency.

5.3 Dependency Representations in Conceptual Graphs

This section describes conceptual graphs as a knowledge representation and shows the basic approach we will be using. Details are given as to how conceptual graphs represent dependencies and in particular how the representation is extensible and flexible. Since this is a pilot study, we do not claim we have completely exhausted all possible dependencies; we do, however, claim that we expect to be able to accommodate any future modifications to our theory and characterization of dependencies in general. This section shows our current thinking.

As a first step in our analysis, we focus upon a much simpler type of dependency, the case of a dependency between only two entities, D(A, B). In the case where A depends upon B and B depends upon A, this dependency can be seen as a bi-directional relationship, Figure 10. We call this bi-directional dependency an interdependency. Given such an interdependency between two entities, we can now separate the dependency D(A, B) into at least two one-way, or unidirectional dependencies, Figure 11, d1(A, B) and d2(B, A), where we shall use lower case for unidirectional dependencies and continue using upper case for bi-directional dependencies. We can be sure that this is always the case, because we have included "independent" in our type hierarchy for dependencies (refer to Figure 10).

5.3.1 High-Level View

This section gives a high-level view of how dependencies are characterized with conceptual graphs. It comprises merely an overview whose details are fleshed out in the next section.

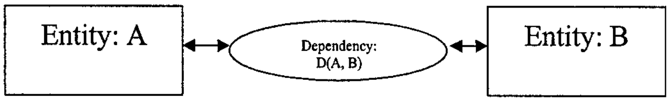


Figure 10. Bi-directional dependency, or interdependency, between two entities

In the simplest case of a dependency, a unidirectional dependency between two entities, d(A, B), we can say that A depends upon B. If A depends upon B, then a change in B implies a potential or possible change in A. As in Keller, *et. al.* [Kel00], we refer to A as the dependent and B as the antecedent. This definition of the simplest form of a dependency is very like the definition of dependency given in [Boo99] and is depicted in Figure 4.

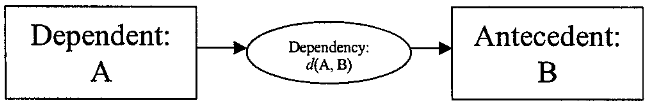


Figure 11. Graphical representation of the simplest dependency

Again, it is important to note that this definition of the simplest case of dependency expresses a one-way direction for the dependency. As Briand, Wust, and Lounis [Bri98] point out, it is not only possible, but common that a bi-directional dependency exists; and, given the definition of the most general form of dependency above, it is also conceivable to have such an interdependency demonstrated between N entities where N>2.

Our initial work is based upon the decomposition of complex dependencies into unidirectional, binary relations. The complex dependency can be broken into some number of unidirectional dependencies. As described above, it is easy to see that in the case of an interdependency between two entities, the bi-directional dependency can be described using at least two one-way dependencies between the two entities. We expect that in a case of symbiosis among N entities, the symbiosis can be represented by at least $2(\ )$ unidirectional dependencies. We use the term "at least" here because there may be multiple types of dependency existing between any two entities. For example, both an intermittent, time-based dependency and a static structural dependency may be involved in the interdependency. Even if the dependency is of a single type, such as a functional dependency, it could include several different and specific "needs" of the entities. In that case, a separate unidirectional dependency could be defined for each specific need. Our continuing research will include a more in-depth investigation of this expectation.

We use conceptual graphs to represent all system-relevant knowledge, including dependencies. The system of conceptual graphs (CGs) as developed by Sowa[Sow84] is a powerful modeling language that has been shown to be useful in domain modeling and in requirements modeling [Del91] [Del92] [Rya93] [Tha98]. A conceptual graph, Figure 12, is made up of concepts (represented by square or rectangular boxes) and relations (represented by circles or ovals) between the concepts. A concept is labeled with a type identifier, an optional referent (indicating a particular individual or set of individuals of that type), and a possible value or measure.

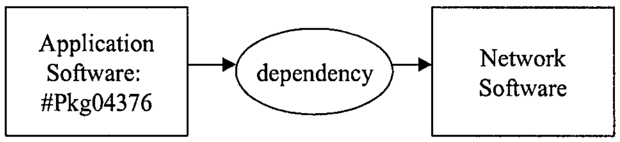


Figure 12: CG for top-level dependency

Relations are connected to concepts with directed arrows where the direction of the arrow is usually determined by linguistic conventions. The ontology of a particular domain can be captured in the type hierarchies that exist for both the concepts and the relations. These type hierarchies represent the subtype/supertype classifications and include a set of definitions of the various types. As in UML, multiple supertypes are allowed, producing a type hierarchy that is not a tree, In Cyc, the types form a lattice [Dav90], which is a partially ordered set with a unique top and bottom, (see Appendix A for description of a CG type and relation lattice). For simplicity, we usually show a type lattice as a hierarchy, with an implicit least element. For a mathematical treatment of lattices and how they support conceptual modeling, see [Gan99].

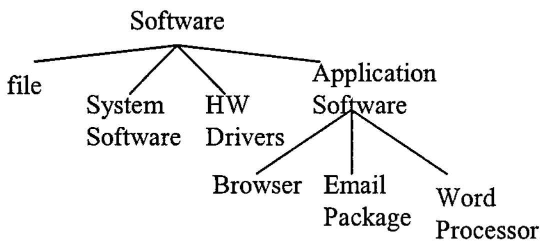


Figure 13: Example concept type hierarchy

Separate type hierarchies are defined for concepts and for relations in conceptual graphs. Figure 14 depicts a portion of our dependency type lattice as an example of a relational type hierarchy. For the complete hierarchy, see Figure 17.

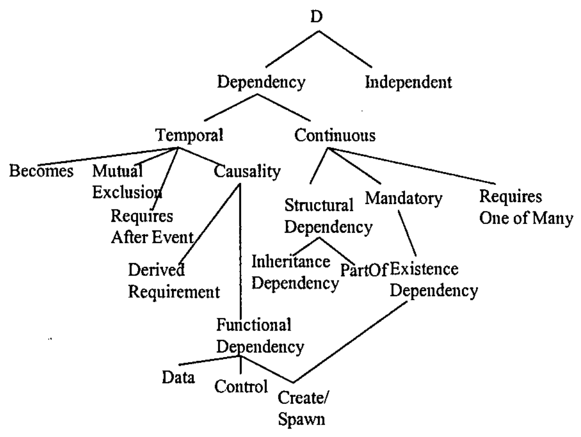


Figure 14: Dependency relation partial type hierarchy

Much research is ongoing concerning the specification of ontologies of objects [Bor96], however the research on relational ontologies is still in the initial stages[Gua00].

5.3.2 Detailed View

Each concept or box in a conceptual graph may be expanded into a complete graph defining the concept. With relational expansion, relations may also be expanded into a more detailed graph that defines the relation, Figure 10. Details of the relations and linked concepts (*e.g.*, Frequency, Need, etc.) will be described in sec. 5.4 Dependency Ontology below. The purpose of this section is to make the point that dependency can be treated at a high level as a primitive, or it may be treated at a lower level with specific attributes and relationships.

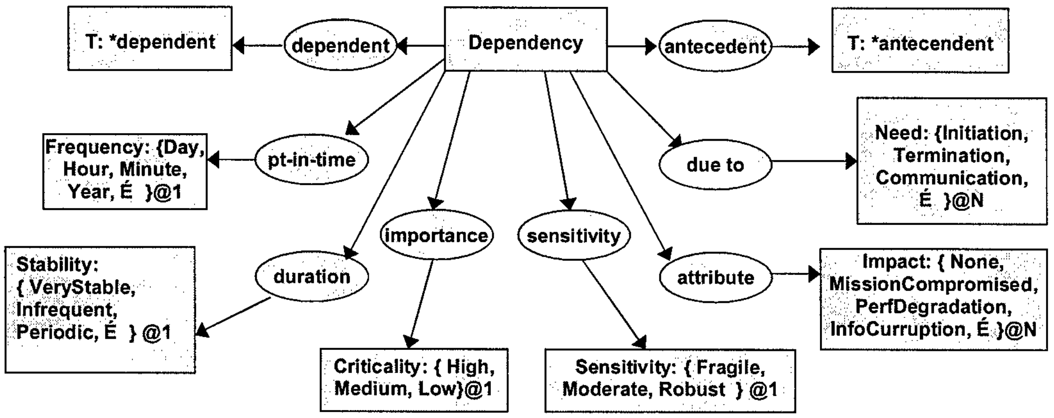

T: *dependent
dependent
Dependency
antecedent
T: *antecendent
Frequency: {Day, Hour, Minute, Year, É }@1
pt-in-time
due to
Need: {Initiation, Termination, Communication, É }@N
importance
sensitivity
Stability: { VeryStable, Infrequent, Periodic, É } @1
duration
attribute
Impact: { None, MissionCompromised, PerfDegradation, InfoCurruption, É }@N
Criticality: { High, Medium, Low}@1
Sensitivity: { Fragile, Moderate, Robust } @1

Figure 15: Relationally expanded CG for dependency

Figure 15 depicts the graph obtained when the dependency relation in Figure 12 is expanded to include the dependency definition (see Appendix A for details on CG relation expansion). Figure 15 also serves as our definition of the **dependency** relation that provides our primary formalism for representing the several dimensions we have identified so far. In this way, conceptual graphs facilitate the type-based definition of the dependencies and allow the modeling of the dependency at a low level of detail (as in Figure 15), or at a higher level of detail (as shown in Figure 12). Both the graphs exist in the knowledge base together; therefore the dependencies can be simultaneously modeled at multiple levels of granularity.

As domain knowledge is accumulated in CG definitions, a set of canonical graphs is developed that represents the domain in the knowledge base. CGs support inferencing, querying, some natural language processing, and knowledge interchange through the conceptual graph Interchange Format (CGIF) which is in the process of becoming an American National Standards Institute (ANSI) standard[Sow00a]. The graphical form of CGs is easy to read in human form and the standard machine representation allows the automation of the translation between CGs and more widely used formats such as UML.

5.4 Dependency Ontology

This section introduces the basic dependency ontology we have developed. It is important to state that most approaches to modeling dependency take some existing software description technique and "add on" some form of dependency representation. Ours is a rooted approach, whereby we have analyzed the fundamental notion of dependency itself. As a general idea in human thought, we have chosen to represent it using a well-known knowledge modeling representation, namely conceptual graphs.

We have chosen conceptual graphs as the language in which to represent dependencies because of the support given to establishing a detailed ontology of relations. As shown above, conceptual graphs allow a representation of relations that is equally as powerful as the representation of concepts or objects. The use of conceptual graphs alone, does not guarantee that the dependencies are captured and completely specified. It is important to add the ontology of dependencies to conceptual graphs and to establish a methodology for specifying the dependencies in a system. We expect that with a well-defined ontology of dependencies to choose from, we will be able to automate more of the requirements specification for a system. The requirements engineer can specifically identify the type or types of dependency involved and can fill in a template for the attributes of those dependencies. The system can then construct graphs at varying levels of detail to model whatever dependencies exist in the system.

Just as it is possible to specify and model the dependencies between objects in the domain, it is also possible to view the artifacts of the Requirements Engineering (RE) process as concepts in a model for which the dependencies can be described. The dependencies between the requirements themselves, can be captured and specified using the same approach, allowing modeling at both the domain and the requirements levels.

Our present work is focusing on the completion of the dependency ontology. The methodology for applying our dependency representation is a topic for future research.

Our approach to defining the ontology, takes a category theory perspective of the notion of "dependency". Using the definition of an orthogonal set of attributes associated with the dependency concept, we can populate the type lattice. More detail about the types defined in the present lattice is given in [Cox01]. Our search is for the definitive set of attributes which will give us every dependency type required by a modeler, and from that we hope to derive a complete type lattice.

Keller, *et. al.* [Kel00] is the only source in which we have found an attempt at the classification of dependencies based upon such attributes. Keller, et. al. [Kel00] lists six attributes of dependency which are represented as orthogonal axes in a six-dimensional dependency space wherein each dependency can be graphed. Our initial set of attributes, which are applicable to all dependencies, includes two attributes from [Kel00], criticality and strength (the latter we call Frequency). However we believe that the other four attributes cited by [Kel00], rather than being associated with the dependency, would be more properly represented as attributes associated with the system components (the entities A and B) or with the system, itself. For example, the "component type" cited by [Kel00], is not an attribute of a dependency as much as it is an attribute of the entity, A, being modeled, and the attribute "dependency formalization" is actually dependent upon the particular system in question.

To the two attributes we have taken from [Kel00], we have added the attributes of impact, sensitivity, stability, and need as important to all dependencies. Keller *et. al.* [Kel00] also addresses the issue of "time", although it is not included in the six-dimensional dependency space. This is very like the attribute we have named stability. Based upon the values of the attributes, we can then establish a hierarchy of dependency types[Gua00]. Our initial attributes are shown with general definitions in Figure 16.

Feature	Definition	Possible Values
Need	The need on which the dependency is based. Is usually represented as a list of required capabilities.	Authorization Resources Provided Testing At lower levels could include Text Editing, Computation, Network Access, File Save/Retrieval, *etc.*
Criticality	A measure of the importance of this dependency to the success of the "needing" entity.	Not Applicable High Medium Low
Frequency	A measure of the frequency of the need/criticality – how often does the need/importance influence operation?	Daily Hourly Yearly A numeric value representing how often the dependency exists during a particular time period.
Impact	Possible repercussions of failure at this dependency.	None Mission Compromised Information Unreliable Performance Degraded Corruption/Loss of Information/Communication
Sensitivity	How vulnerable is this dependency to compromise or failure?	Fragile Moderate Robust
Stability	A measure of the continuity of the dependency's vulnerability to compromise or failure (sensitivity) over time. One way of looking at stability is to ask the question: "When is the dependency fragile/moderate/*etc*?"	Very Stable Infrequent Periodic Certain Defined Times

Figure 16. Dependency Related Concepts.

Figure 17 shows further dependency types. We stress that the actual set of dependency types and attributes is still under investigation; what we present here is meant to suggest the kinds of dependencies we have identified so far. The types are shown as a nested hierarchy.

- Dependency
    - Structural Dependency
        - Sameness
        - Part of
        - SpatiallyBound
    - Temporal Dependency
        - Duration
            - Infinite
            - Interval
            - Event Triggered
                - Event-based Interval (subtype of Interval too)
        - Causal Type
            - Direct Causality
            - Unknown Causality
                - Statistical Dependency
                    - Unknown Correlation
                    - Known Correlation
                        - ProbabilityOne
                        - ProbabilityLessThanOne
    - Conceptual Dependency
        - Logical Dependency
            - Implication
            - Inheritance
        - Requirement
            - Mandatory
            - Existence Dependency
            - Empirical

Figure 17. Dependency Type Hierarchy.

Figure 17 is a succinct summary of our work and deserves some more detailed explanation.

Structural Dependency refers to dependencies of structure. It can be thought of as static dependencies that can be known from static descriptions of components and processes. Sameness is a special case where two things are dependent because they are the same thing; this can sometimes occur in specifications where systems are partitioned and developed separately. Part Of dependencies refer to part-whole relationships, as described in UML composition. Spatially bound are things that are dependent because they are in proximity to each other; for example, two servers are spatially dependent if they are in the same room or on the same electrical circuit.

Temporal Dependency refers to dependencies between components or processes that are dynamic or time-based. Duration refers to temporal dependencies that are duration-based: Infinite (meaning duration is permanent), Interval (where dependency occurs on a regular basis) or Event-Triggered (dependency is due to an event). Event-Triggered differs from a Causal Type which is where an entire component or process causes the dependency. Causal dependencies are where some component or process creates or alters another component or process. Direct causality is where there is a deterministic or mechanical explanation of the alteration; Unknown Causality is where a change is produced, but there is no known explanation of it. Mutual Exclusion is where two or more components or processes cannot exist at the same time (but may each exist separately at different times).

Conceptual Dependency is a general category for dependencies that are no apparent in some temporal or structural sense. **Logical dependency** is a kind of conceptual dependency that is the result of some logical process: e.g., the adage in foreign policy that states "the enemy of my enemy is my friend" makes a logical dependency statement about who is my "friend". **Implication** is the more specific IF-THEN logical rule (not to be confused with the causal IF-THEN which is of type **Causal Type** under **Temporal Dependency**). **Inheritance** is the property of object-oriented systems which is the "kind of" relationship between components. From a programming standpoint, this is sometimes called a structural dependency; however, the notion of objects and their super- and sub- classes are a conceptual notion, not a physical one. The other main kind of conceptual dependency is a **Requirement**, which can be one of three types: mandatory (something dependency that must hold), existence (the existence of one thing requires the existence of another) and empirical (an observed requirement that is neither mandatory nor existence-based).

- Dependency Attribute
  - Cardinality
    - Independent
    - Binary
    - N-ary
- Direction
  - Symmetric
  - Anti-symmetric
  - Asymmetric
- Impact

Figure 18. Dependency Attribute Hierarchy.

5.5 Conceptual Graphs Summary

Much of this report will describe conceptual graphs strengths and weaknesses, but this table summarizes them in relation to our original criteria.

Conceptual Graphs' Dependency Representation	
Use(s)	System requirements and specification, enterprise modeling, process modeling, natural language processing, cooperative environments, etc.
Strength(s)	Inference, constraint modeling, process modeling
Weakness(es)	Knowledge acquisition, support
Potential	With interoperability among other knowledge bases, ability to incorporate pre-existing deep knowledge about domains.

Table 5 Conceptual Graph Representation Capability

This section described how we model dependencies using conceptual graph types and relations, and then outlined the types we have identified as comprising the dependency domain. The strategy is to establish specific relationships between the different kinds of dependencies and their attributes or features. We believe this provides a comprehensive approach with the potential to capture all the needed knowledge about dependencies, and allows that knowledge to be manipulated and analyzed using conceptual graphs. The next section presents examples of how the approach may serve to represent and detect dependency problems.

6. RESULTS AND DISCUSSION

As noted in Section 3, the purpose of this study was to determine the capability and suitability of Conceptual Graph technology to represent dependencies, with a long-term interest to form a basis for a dependency language definition. Further, it was found that little formal research into the meaning of dependencies was available. The following sections present the conclusions reached in this study. Once a definition was developed in Conceptual Graphs with traceability to a common dictionary definition, the capability was established for dependency representations in Conceptual Graphs. Suitability is a more difficult term. We establish suitability by tracing back to previous desired attributes (Section 3), comparing to other methods, and by presenting example uses to establish suitability for dependency representations. While this is clearly not a mathematical proof, demonstration of suitability for a given usage by actual usage is proof of suitability for the examples chosen.

6.1 Attribute Comparisons

This table looks at the two most viable techniques encountered in the study, UML and Conceptual Graphs. For reasons noted in Section 4, Cyc and Bayesian methods were not considered as viable.

Dependency Attributes	Conceptual Graphs	Unified Modeling Language
Traceability	Websters	Unspecified
Quantification	1 – 1, 1 – many, many – many, unspecified number, each, every, distributed, collective cumulative quantification	1 – 1, 1 – many, many - many
Causality	Arrow head in relationships	Arrow head in dependency
Transitivity	Intrinsic in language	For some diagrams, within that diagram type
Composition/ Hierarchical	Intrinsic in language	Multiple diagram types impedes this
Dual Hierarchy	Intrinsic in language	Class/Object diagrams
Temporal	Explicit by user	Sequence and action diagrams
Procedural/Sequential	Explicit by user	Sequence and action diagrams
Contexts	Intrinsic in language	Possible
Logical basis	Formal logic basis	No formal logic basis

Compact	Graphical three symbols used	Graphical multiple symbols
Unambiguous	One diagram type, three symbols	Multiple diagram types, multiple symbols
Standardized	National standard	Limited standard essentially owned by private company
Low Total Cost	Limited texts and training available, less well known	Well known technique, multiple tools, common experience, many sources of training and texts

Table 6 Suitability Comparison

Bottom line in the comparison is that the great strength in Conceptual Graphs is the simplicity of the symbol set and the diagram type. This is also a weakness in that the user must learn how to represent time, sequences, control etc in conceptual graphs. UML has the advantage of using multiple diagram types to model each aspect of a system. This permits each diagram to be tuned to each aspect of a problem. The difficulties are that a user must learn and master multiple types and their appropriateness and symbology. Further, the relationship of one type of diagram to another is defined only in vague senses, with the idea that each representation is more or less normal to the others, so why worry about transforming? This makes automated analysis tools, for dependencies, a much larger problem to develop. Performing inferences and queries becomes virtually impossible.

6.2 Examples Usage

This section provides some illustrative examples of how the conceptual graphs approach works to produce useful dependency analysis. These samples demonstrate the capability to represent the following dependencies:

- Single dependency
- Double dependency
- Join dependency
- Generalization and specialization dependency
- Composed dependency

6.2.1 Dependency Example: Software/System Components Domain

This example assumes the use of a typical PC operating system. Similar examples may be constructed using other operating systems. Figure 19 suggests a simple block diagram of a typical layered operating system architecture, where upper layers make use of services in the lower layers. Software components are shown within the dashed lines.

As depicted in Figure 19, the Browser component is dependent upon services or capabilities of the operating system, including the modem software and the Local Area Network (LAN) software. This simple example illustrates a case where many dependencies are all of one kind; that is, components depend on the prior initiation of their antecedents. In this example, we also assume some executables for the browser reside on a server reached via the LAN.

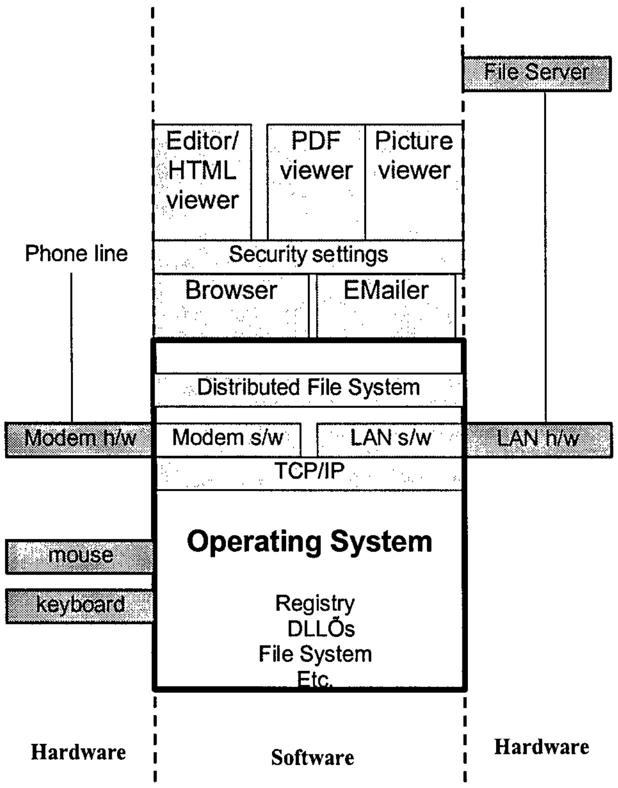


Figure 19. Browser Example.

The Browser may be started manually, when the user clicks on a shortcut icon or when the user runs the executable from the command line. The Browser may also be started automatically from the user's startup folder. The Browser is therefore dependent upon either the user or the startup folder for initiation, Figure 20.

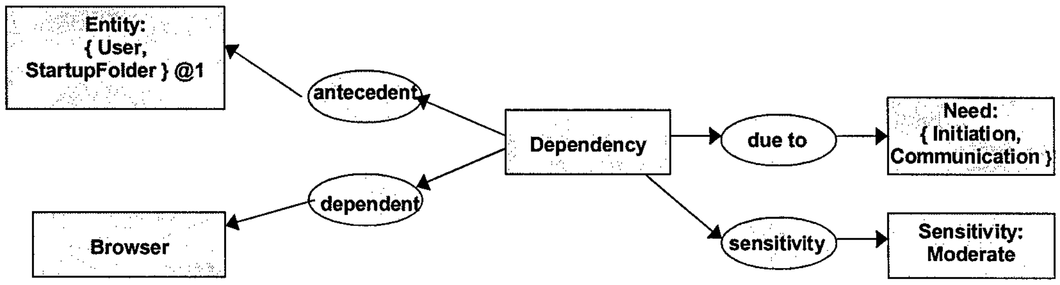


Figure 20. Initiation Dependency.

In this situation, the antecedent is an instance of one component from the set of three given (see Appendix for set notation). The Browser is dependent upon an initiator that can be

1) The user via a shortcut icon, or command line, or

2) The startup folder

This dependency with attribute [Need: { Initiation, Communication }] is of type "control dependency" which is a subtype of "functional dependency". It also serves to illustrate how the conceptual graph notation handles the notion of choosing among a set.

The Browser's dependency on the server is of a different kind. Upon initiation, the executable code for the Browser must be fetched by the operating system from the server on the LAN, Figure 19. Therefore the Browser is also dependent upon all network software and hardware interfacing the particular system to the server in order to begin to run. This dependency is still a functional dependency, but it is not a control dependency. It is a data dependency wherein the data required is the set of executable files on the server. A data dependency can be of either the read or the write types. This dependency is of the read type and can be called an access dependency. It is not an initiation dependency, because the dependency comes into existence after the browser is already initiated. It is nonetheless an access dependency (actually a set of dependencies) that must be satisfied before the browser can run.

A key aspect of using conceptual graphs to analyze dependencies is that conceptual graphs already support operations to combine, project, and otherwise transform or match graphs. This gives us the capability (for free, as it were) of deriving composite dependencies, simply by combining graphs of the individual dependencies. No new conceptual graph operations or transformation rules are required.

Using the strategy outlined above, dependency graphs can be drawn for each of the dependencies described here. Figure 21 shows a set of conceptual graphs representing the individual dependencies of the previous example. Using conventional conceptual graph operations, we can obtain a combined dependency, which automatically captures the various dependency chains inherent in the descriptions, which will be shown after the separate graphs.

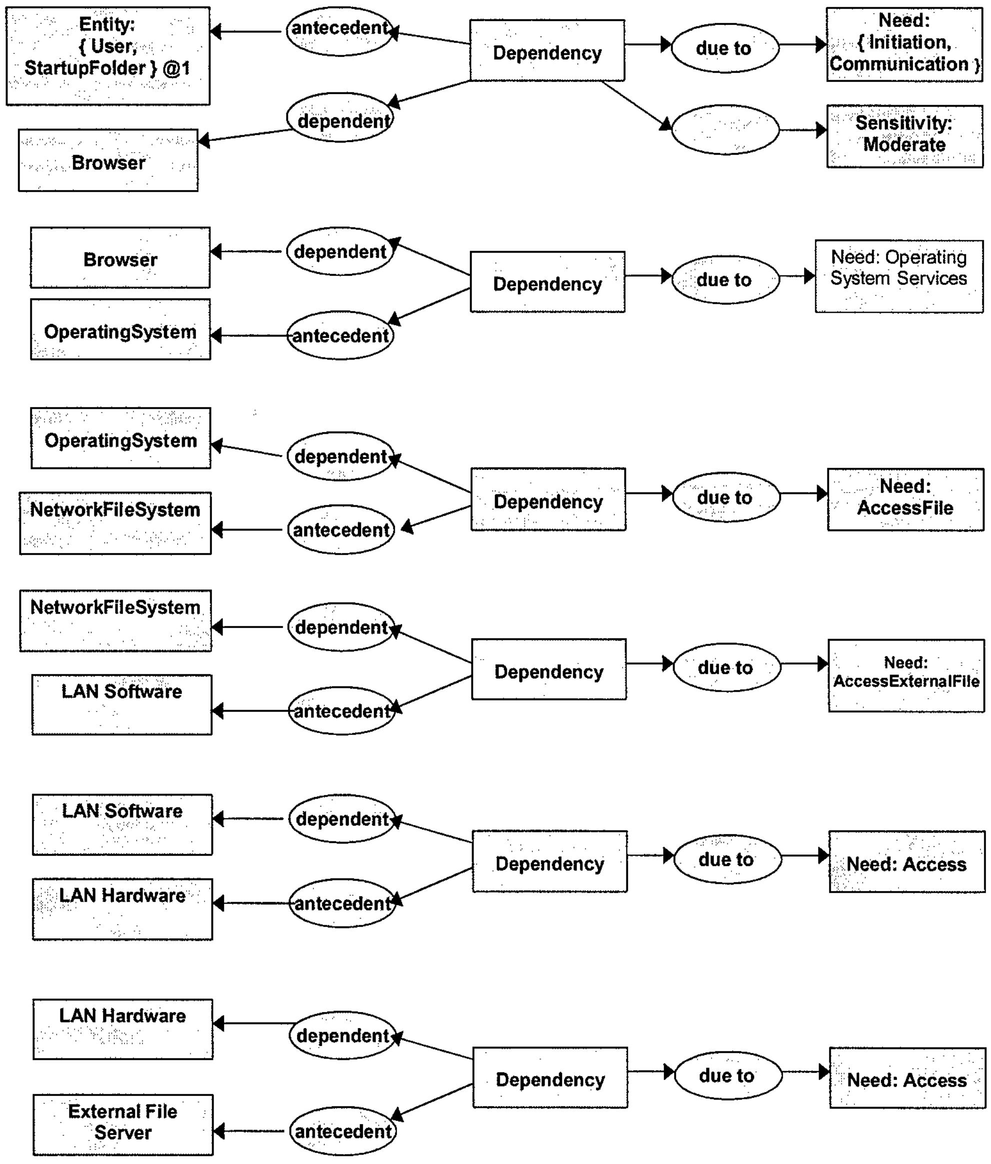
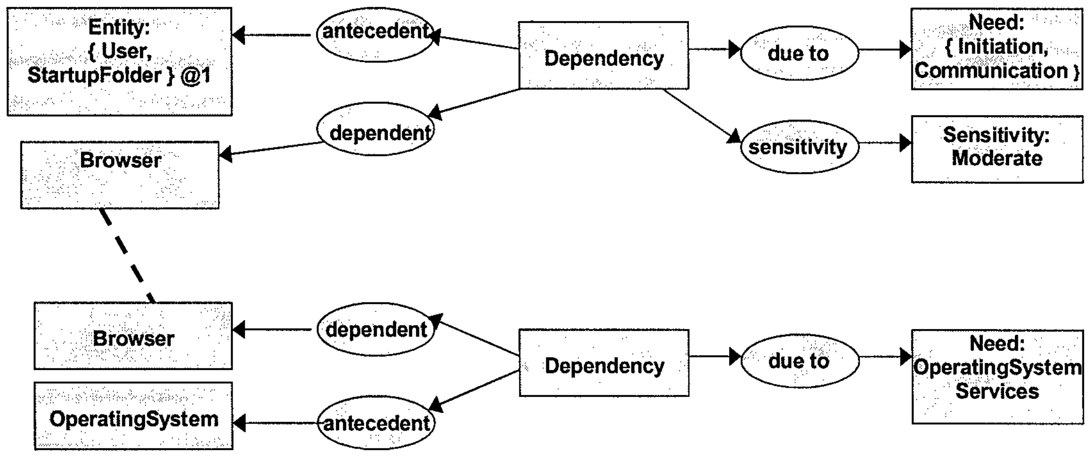
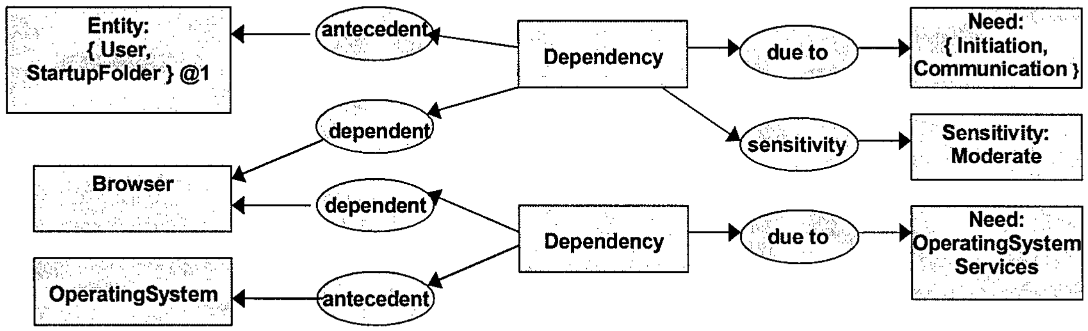


Figure 21. Individual Dependencies To Be Combined.

Using CG join operations, the chain of dependencies given in Figure 24 can be produced. We show here one part of the derivation; the rest of the graphs can be combined in a similar fashion to obtain Figure 24. Consider the top two graphs in Figure 21. A join can be performed in several ways; e.g., two [Need: Access] concepts with the same referent can be joined, in which case we would be making an inference about access needs. Generally joins are performed around some specified concept type(s); in this case, the types would be entities related to dependencies. The [Browser] concept s can be considered the same individual, as we illustrate in Figure 22(a); therefore the two graphs can be joined to obtain the graph in Figure 22(b). Note the dashed line in Figure 22(a): it represents a line of identity whereby two distinct concepts are intended to mean the same individual; in this case, the same individual Browser. We say that the two graphs in Figure 22(a) can be joined around the [Browser] concepts, replacing the two occurrences with just one, thus joining the two graphs together to form the larger graph as shown in Figure 22(b).



(a)



(b)

Figure 22. Joined Graphs.

Assuming that expanded dependency graphs can be drawn for the each of the other dependencies listed in this example, a CG system will be able to quickly join the graphs to form a complete picture of the dependencies, as shown in Figure 23.

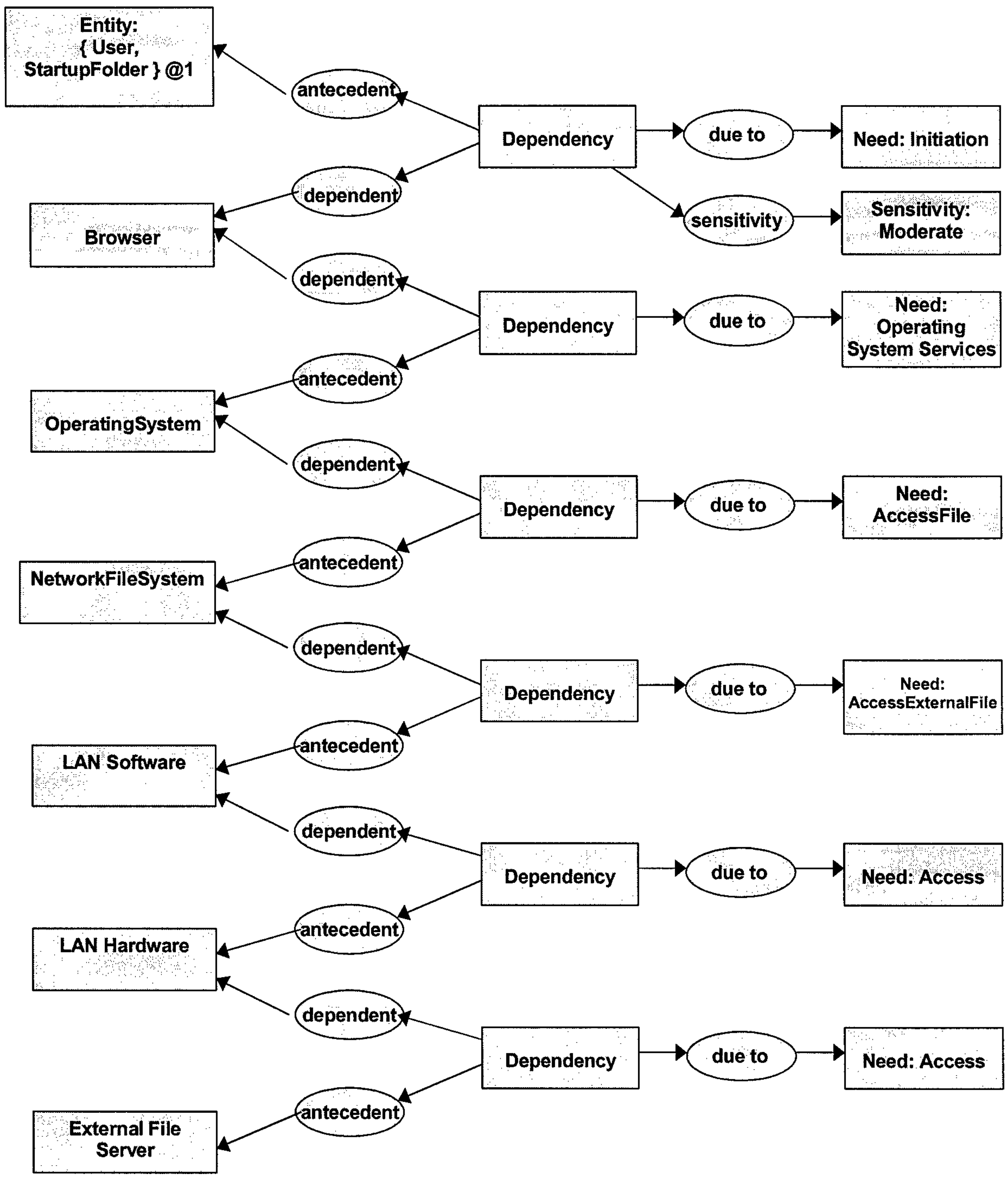


Figure 23. External File System Dependency Representation (Detail Level).

Once the executable for the Browser has been successfully fetched from the external file server, the Browser can then resume execution. The execution is, of course, dependent upon the O.S. As the Browser is executing, it depends upon the keyboard and the mouse (via the O.S.) for interaction with the user. These input devices allow the functional control dependencies to be satisfied. As a result of these inputs, the Browser uses an Editor or Hyper Text Markup Language (HTML) Viewer such as MS Word, a picture viewer, and possibly a Portable Document Format

(PDF) viewer to decode and display the web pages to the user. As part of the interaction between user and web page, it is common for the user to be allowed to click upon a link in a web page that evokes the default email program. These are common dependencies that come into being at various times during the operation of the browser.

Let us look more closely at the dependency between the Browser and the Editor/HTML viewer. The default editor is usually established in the Browser settings. If Microsoft Word is that default editor, but for some reason Word is found to be compromised or failed for some reason, the Browser may then be told by the user to use a different editor and continue to operate. In that case, a new editor may be used to fulfill the "need" that had previously been filled by Microsoft Word.

An unstated dependency also exists between the various Browser software packages. If a system has Netscape, America OnLine (AOL), and Microsoft Explorer, trying to use any two at the same time can produce all kinds of havoc. There is therefore a mutually exclusive dependency between the three packages.

6.2.2 Abstraction Example: Contracting Dependencies to Higher Level

One power of our approach is that the detailed level is not the only way to look at the dependencies. The conceptual graph approach supports both a high level and a detailed view. These dependencies can be expressed as a chain of transitive access dependencies. This is accomplished through a definition contraction, which is the inverse of the relation expansion rule. This allows us to remove the detailed features of the dependency relations and simply write the relationship as a (dependency) relation between the antecedent and the dependent, to obtain a graph as in

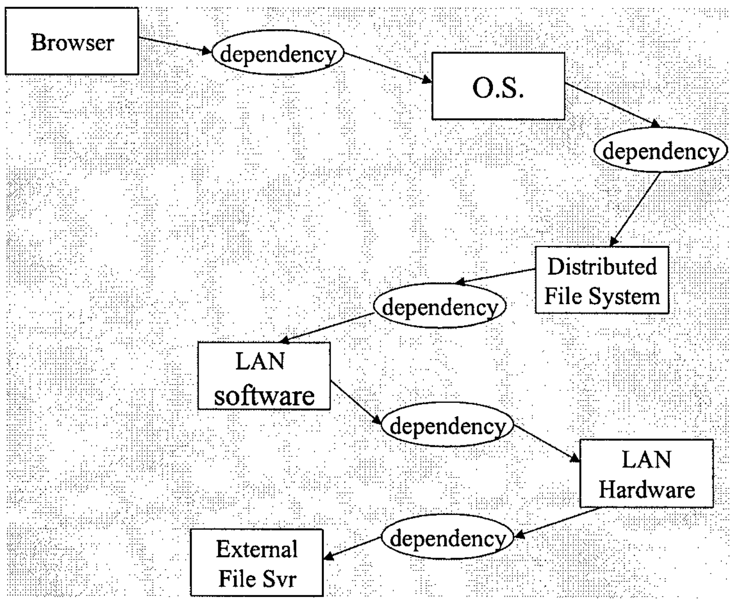


Figure 24. External File System Dependency Representation (High Level).

Further higher-level descriptions could be obtained by using a transitivity rule, which is specified as a regular CG system rule (see Appendix A for description of rule) as in Figure 25. The rule states that **if there is some entity *a* which is the dependent of a dependency relation whose antecedent is *b*, and there is some entity *b*, which is the dependent of a dependency whose antecedent is *c*, then there is a dependency relation where *a* is the dependency of *c*.** This rule allows us to rewrite detail level dependency concepts as higher-level dependency relations.

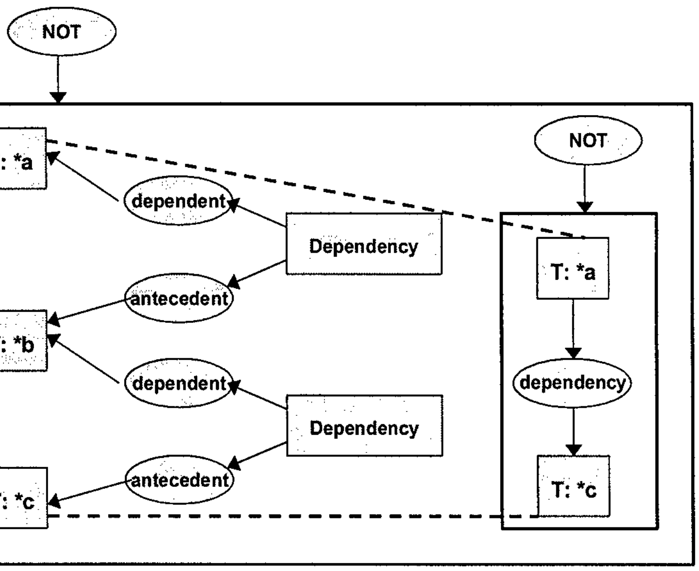
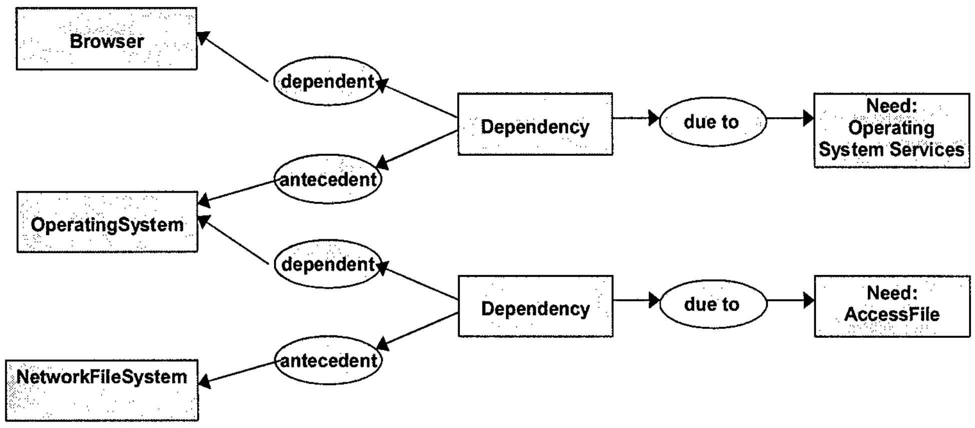


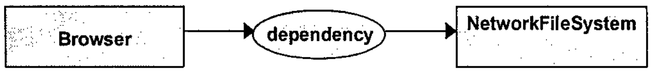
Figure 25. Rule For Dependency Relation Contraction.

It should be noted that because we are using typed dependencies, all dependencies are not necessarily transitive with respect to each other, since they may involve different dimensions or attributes of dependency. Assuming all these dependencies have only been described using the dependency relation, we can draw the graph shown in.

As an example, consider the graph portion shown in Figure 26(a). Using the rule of Figure 25, the graph portion can be rewritten, substituting a **dependency** relation for the detailed [Dependency] concepts and their relations, resulting in the graph of Figure 26(b). Note that CG operations allow the partial matching of graphs, so that the [Need] concepts are transparent under the transformation.



(a)



(b)

Figure 26. Graph Portion For Definition Contraction.

Using the rule from Figure 26(a), the combined graph from Figure 23 can be redrawn at a higher level as in Figure 27. For some purposes, this level would be useful; several approaches (including UML) can represent the same information.

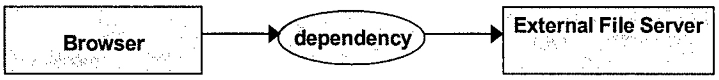


Figure 27. Fully Contracted Dependency Relation.

This example showed how dependencies are represented in the conceptual graph approach, how they are manipulated and how they can be transformed in various ways to suit different analysis needs.

6.2.3 Deeper Knowledge Example: Composable Trojan Horse

If an undetected infiltration through an earlier email containing an Excel attachment introduces part of a Trojan horse program into a system, it can lie dormant until it happens to be triggered by the reception of an attachment containing another part of the Trojan horse. The model shown in Figure 28 illustrates how this would be represented using conceptual graphs. Note the use of co-referent links (or lines of identity) that allow multiple statements to be made about individual components. For example, the email program is dependent on Microsoft Excel and Microsoft Word, and it also is part of the dependencies' need for that program to use the attachment which make up the Trojan Horse. In this example, the Trojan Horse concept represents a potential threat to the system, and the conceptual graph model can help identify the dependencies that support the threat.

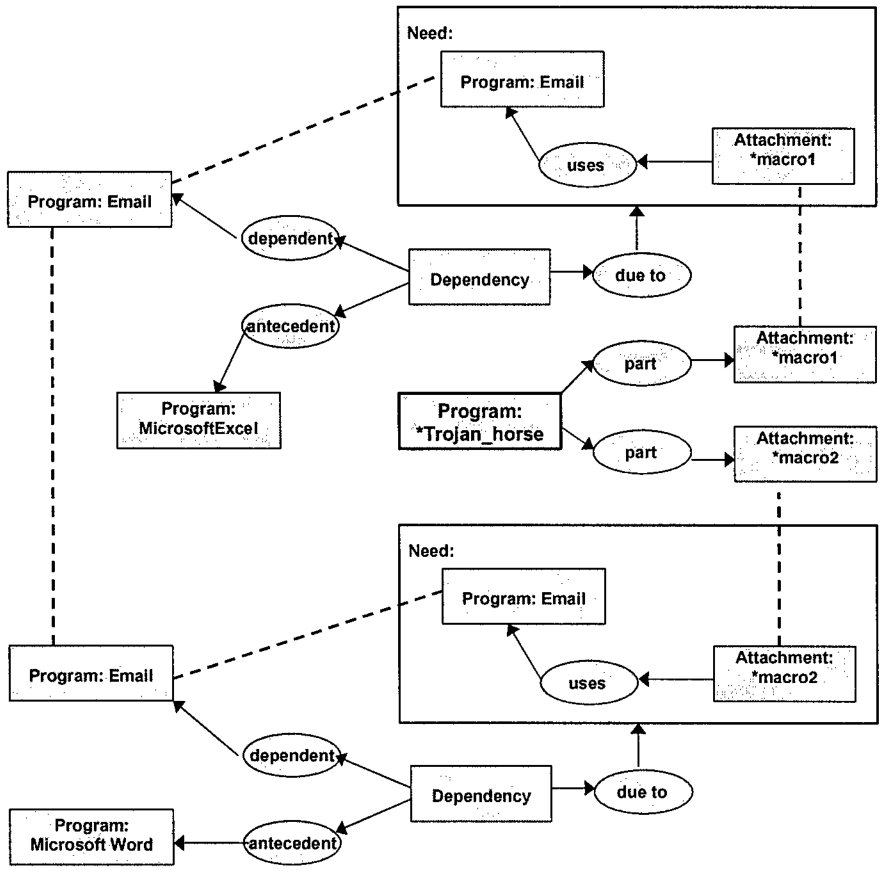


Figure 28. Composable Dependencies.

This example shows more of the expressive power of conceptual graphs: a need can be shown with more complexity and with relationships to other things being modeled, *i.e.* situations or contexts.

6.3 Benefits of Approach

This section summarizes the benefits of the conceptual graph approach to dependency, as described in section 3.

6.3.1 Requirements-Based (Pre-Design) Analysis

Because there is a body of work modeling requirements using conceptual graphs, dependency analysis can be seamlessly performed using existing technologies and techniques. It is clearly beneficial to understand dependencies in a system before it is built, to eliminate or mitigate dependency problems that are revealed. Because requirements decomposition and derived requirements can be represented in the dependency model of the requirements, it is then possible to provide requirements traceability as part of the automated analysis.

6.3.2 Incorporation of Domain Knowledge

If both domain knowledge and the emerging requirements are modeled using this system, it becomes possible to combine the conceptual graphs from the domain modeling with those from the requirements modeling into a single model and to perform analysis of the requirements in comparison to the domain. This gives an added level of requirements checking capability where we can better automate the checking of the correctness of the requirements in direct relationship to the domain knowledge. By contrast, UML offers some limited consistency checking, e.g., a term's usage as noun or verb in appropriate places.

In addition, the CG system adds flexibility as the requirements or the domain changes. Its use facilitates the addition of new domain information and new requirements as they become available. Using the CGs, we can also automate the determination of the impact of each change, showing through the dependency graph, which assumptions about the domain and which previously established requirements are brought into question. After discrepancies are resolved, the new set of requirements can again be automatically translated into the preferred representation of the team members. Since UML does not embody a general semantic model, opportunities for such checking are limited.

6.3.3 Automated Analysis Capability

In addition to the automation gained by applying theorem-proving technology to the graphs, it is also possible to automate such translation based upon the differing views and needs of team members. Since Conceptual Graphs are both a formal representation and a semantic one, automated analysis can detect inconsistencies, and some incompleteness in parts of a specification. It also allows improved automation of both the dependency analysis and the translation into other representations. Such a capability in dependency analysis is comparable to automated testing capabilities in software development. UML is only equipped to handle comparisons with other UML models; as a general knowledge representation, CGs can be translated to/from a variety of other modeling techniques.

6.3.4 Easy Visual Interpretation

The use of conceptual graphs provides an easy to understand, graphically based language in which to express a rich set of dependencies. As can be seen in the previous figures, conceptual graphs express the objects and relationships of a domain in a way that is easily communicated to those outside the computer science field, easing the communication with domain experts. UML also possesses a strong visual interpretation; in fact, so strong that it often lacks an inherent semantics, since terms and words are freely interpreted by the UML diagrams' viewers.

6.3.5 Interoperability To/From Other Systems

Once the domain knowledge or requirements are captured in CG form, the CGIF format allows the translation of the domain knowledge to the representation preferred by the development team. The existence of the CGIF standard will facilitate interoperability and ensure consistency between systems. As additional knowledge bases are constructed with CGs, analysis of dependencies in a rich knowledge environment will be possible. Since CGs are also translatable to other representations (including UML), we can obtain the power of all those other representations and their systems as well.

It is possible that a hybrid approach, for example, combining techniques and tools of UML with conceptual graphs. Such a hybrid approach can always be attractive; we have addressed some issues in UML and believe such a hybrid may be useful. Further work will be required to address this beyond our current SOW.

6.3.6 Extensibility of Dependency Representation

The dependency representation shown in section 3 comprises our initial pilot study in this area. We freely allow for modification and tailoring of the approach as dependency analysis becomes better understood. Since we have developed an ontology covering most aspects of dependency that we can identify, we are confident that additional aspects can be easily incorporated into the ontology. UML's extensibility is merely syntactic; additional stereotypes and constructs can be "asserted" without definition.

6.4 Transitivity Issues

Most approaches to dependency analysis rely on some form of transitivity, namely if A depends on B and B depends on C, then A depends on C. If we were dealing with dependency as a simple directed arc, then we could assert such dependencies as "dependency chains" whereby a succession of transitive dependencies are strung together. This idea is quite old: consider the old motto that goes "For want of a shoe, the horse was lost; for want of a horse, the battle was lost." This is the kind of dependency that is easy to model, if we merely want to discover whether a given component is dependent on another component in general.

In our approach, because dependency has several dimensions, transitivity is much more interesting. Instead of merely saying two things are dependent on each other (either directly or indirectly through a transitivity chain), we are interested in the qualities of the dependency. Along any given dimension (e.g., criticality), we have the choice to consider that dimension's dependency values as transitive. For example, if A has criticality "High" to B and B has criticality "High" to C, then A has criticality "High" to C. We cannot, however, just assume such transitivity in all cases. For example if A has criticality "High" to B and B has criticality "Low" to C, then what is A's criticality to C? Along a dimension, we will need to establish appropriate reasoning techniques to deal with these issues.

When more than one dimension is considered, more issues are apparent. For example, if A has criticality "High" to B and B has frequency "Daily" to C, what can we say about A's dependency on C? In most cases, if the criticality and frequency dimensions are indeed orthogonal, we would not assume transitivity among multiple dimensions. Although this may complicate the analysis, we believe it more accurately reflects the true impact of dependencies.

7. CONCLUSIONS AND SUGGESTIONS FOR FUTURE WORK

7.1 Review

Research on previous work depicting the philosophical and mathematical nature of dependency is nearing completion. Initial attributes to be used in the dependency ontology and an initial type hierarchy have been defined. The conceptual graph representation for the dependency ontology has been defined. Research is continuing with respect to the identification of the best set of dependency attributes, (and therefore the best dependency ontology), an algebra of dependencies, and the definition of the method of application of the dependency algebra to specific modeling problems. We consider a dependency algebra to be that set of rules which govern the transforming and combining of elements in the language. Since CG's already have a well-established set of inference and transformation rules [Sow1984], we intend to rely on those existing rules as our algebra for manipulating expressions in the language.

The following publication was a direct result of this work. A copy is attached.

Cox, Lisa and Delugach, Harry, "Dependency Analysis Using Conceptual Graphs," *Supplementary Proc. 9th Intl Conf. On Conceptual Structures* (ICCS 2001), Palo Alto, CA, July 31-Aug. 3, 2001, pp. 117-130. The paper can be accessed through its online archive:

`SunSITE.Informatik.RWTH-Aachen.DE/Publications/CEUR-WS/Vol-41/`

We briefly evaluate some of the findings of our research.

7.2 Development Process

We will develop a unified methodology and analysis environment incorporating multiple heterogeneous existing technologies for analyzing dependencies in complex distributed computer system. The system will allow a user to identify and understand a system's vulnerabilities that are caused by interactions between dependent components.

Since complex distributed computer systems are composed of large numbers of interconnected heterogeneous components from multiple vendors, compromise of any component may potentially affect many other components in subtle, unexpected, and unacceptable ways. Dependency analysis seeks to uncover and highlight all dependencies within a system and to provide a measure regarding the degree of dependency and of the seriousness of a compromise of a component in its effect on other components. Current dependency analysis techniques lack the rigorous overall semantic content needed to describe these complex dependencies in an adequate manner and do not permit precise statements to be made concerning the effects of compromise. Our current work has demonstrated that a formal characterization of dependency concepts is feasible. The objective of the work advocated in this report is to extend and broaden our formulation of a general dependency representation based on conceptual graphs (CGs) and stored in Conceptual Graph Interchange Format (CGIF). We will develop the ability to *predict* the effect of a component's compromise on other components and on the system as a whole. We will develop the capability to present these predictions using any one of several current system analysis methodologies.

Our approach will encompass other approaches by exploiting and consolidating existing technologies and methodologies and translating any dependencies from their representation based on other techniques (e.g., use cases, fault trees, statistical methods) into conceptual graphs (a graphical, logic-based, semantic network knowledge representation). We will then combine these results into a composite knowledge base. Conceptual graphs will form the common means for expressing the dependencies uncovered in each technique. Once in conceptual graph form, the knowledge base can be analyzed and mined based on mission plans intrusion scenarios, etc, additional dependencies can be discovered, and the seriousness of a compromise can then be determined. Results of the analyses can then be presented back to users in any form the user chooses by translating from the conceptual graph form into the desired format.

As an initial system for study, we will be looking at our university's information technology infrastructure, including its fiber-optic switches and software, routers, user authentication, and component integration procedures. We plan to work with DARPA to select a second system for study, should time and resources permit.

The research plan will be as follows.

- Develop a rich and complete ontology of dependency concepts, including precise definitions, a type hierarchy and classification criteria to establish a formal basis for dependency analysis.

- Develop an essential dependency algebra – to include rules of inference in conceptual graphs that deal with dependency concepts and relationships – so that a common framework can be used to analyze dependencies from many sources and infer additional dependencies.

- Research other existing dependency analysis methodologies/representations and develop translation methodologies and algorithms, showing how the conceptual graph, knowledge-based approach can subsume, consolidate and exploit their semantics.

- Perform analysis on a system description to validate the methodology and establish an architecture for a production-level analysis system. This activity will lay the groundwork for a workable system, address usability issues, and show a complete example.

- Perform a set of experiments using the methodology to validate the techniques and methodologies developed and to ensure complete coverage of the dependency description universe.

- Provide education/introduction in using these techniques to other information assurance researchers and scientists, so that our methodology will be applied and interpreted in meaningful ways.

- Develop an explanation facility that will present conceptual graph analyses in any of the originating methodologies' representations, both to help in validating the correctness of our methodology and to assist developers in understanding their system's vulnerabilities.

- Perform dependency analysis on medium-scale systems that employ clearly disparate levels of detail in their multiple description models. This step will address scalability issues as well as "zooming" issues.

- Develop a "self-healing" analysis sub-system that can automatically detect, understand and adapt to faults, based on actual system operating behavior, so that systems are continuously being analyzed for new faults.

- Develop initial learning strategies and techniques based on dependency knowledge to guide humans in the effective development of new systems, since the best way to reduce vulnerabilities is to prevent them from being introduced during system development.

- Deploy a production-grade system in an actual environment, to test performance and operational issues.

The long-term benefits of such efforts are: better security in software integration and a better detection and understanding of fault causes. This includes intrusion detection and component failure. By providing a detailed dependency profile for a new component, developers will know where to focus testing and security risk mitigation efforts. By obtaining explanations of (possibly complicated) dependency vulnerabilities, system users can detect and understand fault causes. Real-time and/or "self-healing" strategies will make possible some near-real-time automatic adaptation by a system to compensate for a compromised or failed component.

REFERENCES

1. [Ang97] Angelova, G. Damyanova, S. Toutanova, K. and Bontcheva, K. (1997) "Menu-Based Interfaces to Conceptual Graphs: The CGLex Approach," in *Conceptual Structures: Fulfilling Peirce's Dream*, vol. 1257, Lecture Notes in Artificial Intelligence Series, D. Lukose, H. S. Delugach, M. Keeler, L. Searle, and J. F. Sowa, Eds: Springer-Verlag, Berlin, Germany pp. 603-606.

2. [Ang00] Angelova, G. Nenkova, A. Boycheva, S. and Nikolov, T. (2000)"Conceptual Graphs as a Knowledge Representation Core in a Complex Language Learning Environment," in *Working with Conceptual Structures: Contributions to ICCS 2000*, G. Stumme, Ed. Aachen,: Springer - Verlag, Berlin, Germany, pp. 45-58.

3. [Bal00] Baldwin, J. F. Martin, T. P. and Tzanavari, A. (2000)"User Modeling Using Conceptual Graphs for Intelligent Agents," in *Conceptual Structures: Logical, Linguistic and Computational Issues*, vol. 1867, Lecture Notes in Artificial Intelligence, B. Ganter and G. W. Mineau, Eds: Springer-Verlag, Berlin, Germany pp. 193-206.

4. [Bev00] Bevilacqua, Andrew T. (2000) *Cognitive Reasoning Engine Toolkit User's Manual*, Bevilacqua Research Corporation, Huntsville, AL.

4. [Boo99] Booch, G. Jacobson, I. and Rumbaugh, J. (1999) *The Unified Modeling Language User Guide*, 1st ed: Addison-Wesley Reading, MA.

5. [Bor96] Borgo, S. Guarino, N. and Masolo, C. (1996) "Stratified Ontologies: The Case of Physical Objects," presented at European Conference on Artificial Intelligence Workshop on Ontological Engineering, Budapest, Hungary.

5. [Bos97] Bos, C. Botella, B. and Vanheeghe, P. (1997) "Modeling and Simulating Human Behaviors with Conceptual Graphs," in *Conceptual Structures: Fulfilling Peirce's Dream*, vol. 1257, Lecture Notes in Artificial Intelligence, D. Lukose, H. S. Delugach, M. Keeler, L. Searle, and J. F. Sowa, Eds.: Springer-Verlag, Berlin, Germany, pp. 275-289.

6. [Bri98] Briand, L. C. Wust, J. and Lounis, H. (1998) "Using Coupling Measurement for Impact Analysis in Object Oriented Systems," presented at IEEE International Conference on Software Maintenance (ICSM98), Bethesda, MD.

7. [Car94] Carbonneill B. and Haemmerle, O. (1994) "Standardizing and Inferfacing Relational Databases Using Conceptual Graphs," in *Conceptual Structures: Current Practices*, vol. 835, Lecture Notes in Artificial Intelligence, W. M. Tepfenhart, J. P. Dick, and J. F. Sowa, Eds.: Springer Verlag, Berlin, Germany, pp. 311-330.

8. [Che97] Chein, M. (1997) "The CORALI Project: From Conceptual Graphs to Conceptual Graphs via Labeled Graphs," in *Conceptual Structures: Fulfilling Peirce's Dream*, vol. 1257, Lecture Notes in Artificial Intelligence, D. Lukose, H. S. Delugach, M. Keeler, L. Searle, and J. F. Sowa, Eds.: Springer-Verlag, Berlin, Germany, pp. 65-79.

9. [Che98] Chein M. and Mugnier, M. L. (1998) *Conceptual Structures*, Lecture Notes In Artificial Intelligence, vol. 1480. Montpelier, France: Springer-Verlag, Berlin, Germany.

10. [Che97] Chelba, C. Engle, D. Jelinek, F. Jimenez, V. Khudanpur, S. Mangu, L. Printz, H. Ristad, E. Rosenfeld, R. Stolcke, A. and Wu, D. (1997) "Dependency language modeling," Center for Language and Speech Processing, Johns Hopkins University, *1996 Large Vocabulary Continuous Speech Recognition Summer Research Workshop Technical Reports*, Research Note 24, April 15.

11. [Che76] Chen, P. S. (1976) "The Entity-Relationship Model - Toward a Unified View of Data," *ACM Transactions on Database Systems*, vol. 1, pp. 9-36.

12. [Cox01] Cox, L. C. Delugach, H. S. and Skipper, D. J. (2001) "Dependency Analysis Using Conceptual Graphs," presented at *International Conference on Conceptual Structures 2001 ICCS2001.*

13. [Cre00] Crestana-Jensen V. M. and Lee, A. J. (2000) "Consistent Schema Version Removal: An Optimization Technique for Object Oriented Views," *IEEE Transactions on Knowledge and Data Engineering*, vol. 12, pp. 261-280.

14. [Dav90] Davey B. A. and Priestley, H. A. (1990) *Introduction to Lattices and Order.* Cambridge University Press, Cambridge, U.K.

15. [Del91] Delugach, H. S. (1991) "A Multiple Viewed Approach to Software Requirements," Ph.D. dissertation in Computer Science. University of Virginia, Charlottesville, VA.

16. [Del92] Delugach H. S. and Hinke, T. H. (1992) "AERIE: Database Inference Modeling and Detection Using Conceptual Graphs," in *Conceptual Structures: Theory and Implementation*, vol. 754, Lecture Notes In Artificial Intelligence, H. D. Pfeiffer and T. E. Nagle, Eds. Spinger-Verlag, Berlin, Germany, pp. 206-215.

17. [Del00a] Delugach, H. S. CharGer User's Manual, http://www.cs.uah.edu/~delugach/CharGer.

18. [Del00b] Delugach H. S. and Lampkin, B. (2000)"Troika: Using Grids, Lattices and Graphs in Knowledge Acquisition," in *Working with Conceptual Structures: Contributions to ICCS 2000,* G. Stumme, Ed. Aachen,: Springer - Verlag, Berlin, Germany, pp. 201-214.

19. [Del01] Delugach H. S. and Stumme, G. (2001) *Conceptual Structures: Broadening the Base*, Lecture Notes in Artificial Intelligence, vol. 2120: Springer-Verlag, Berlin, Germany.

20. [Del02] Delugach, H. S. (2002) *Conceptual Graphs: A Primer*, (book in preparation).

21. [deM97] de Moor, A. (1997) "Applying Conceptual Graph Theory to the User-Driven Specification of Network Information Systems," in *Conceptual Structures: Fulfilling Peirce's Dream*, vol. 1257, Lecture Notes in Artificial Intelligence, D. Lukose, H. S. Delugach, M. Keeler, L. Searle, and J. F. Sowa, Eds.: Springer-Verlag, Berlin, Germany, pp. 536-550.

22. [El95] Ellis, G. (1995) "Compiling Conceptual Graphs," *IEEE Transactions on Knowledge and Data Engineering*, vol. 7, pp. 68-81,.

23. [Gan99] Ganter B. and Wille, R. (1999) *Formal Concept Analysis: Mathematical Foundations.*: Springer-Verlag, Heidelberg, Germany.

24. [Gar97] Garner, B. Tsui, E. and Lukose, D. (1997) "Deakin Toolset: Conceptual Graphs Based Knowledge Acquisition Management, and Processing Tools," in *Conceptual Structures: Fulfilling Peirce's Dream*, vol. 1257, Lecture Notes in Artificial Intelligence, D. Lukose, H. S. Delugach, M. Keeler, L. Searle, and J. F. Sowa, Eds.: Springer-Verlag, , Berlin, Germany, pp. 589-593.

25. [Gen97] Genest D. and Chein, M. (1997) "An Experiment in Document Retrieval Using Conceptual Graphs," in *Conceptual Structures: Fulfilling Peirce's Dream*, vol. 1257, Lecture Notes in Artificial Intelligence, D. Lukose, H. S. Delugach, M. Keeler, L. Searle, and J. F. Sowa, Eds.: Springer-Verlag, Berlin, Germany, pp. 489-504.

26. [Gua00] Guarino, N. (2000) "A Formal Ontology of Properties," presented at *12th International Conference on Knowledge Engineering and Knowledge Management*, Juan-les-Pins, France,.

27. [Gua00] Guarino N. and Welty, C. (2000) "Ontological Analysis of Taxonomic Relationships," presented at *International Conference on Conceptual Modeling* (ER 2000).

28. [Hal97] Hall, R. S. D. Heimbigner, and Wolf, A. L. (1997) "Software deployment languages and schema," Dept. of Computer Science, University of Colorado CU-SERL-203-97, http://www.cs.colorado.edu/users/rickhall/deployment/SchemaPaper/Schema.html, December 18.

29. [Han99] Hansen, W. J. (1999) "Deployment Descriptions in a World of COTS and Open Source," Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA http://www.sei.cmu.edu/staff/wjh/DeployDesc.html.

30. [Hay94] Hayes, P. (1994) "Aristotelian and Platonic Views of Knowledge Representations," presented at *Second International Conference on Conceptual Structures* (ICCS94), College Park, MD.

31. [Kel00] Keller, A. Blumenthal, U. and Kar, G. (2000) "Classification and Computation of Dependencies for Distributed Management," *Proceedings of the Fifth International Conference on Computers and Communications* (ISCC 2000).

32. [Lee00] Lee Y. H. and Cheng, A. M. K. (2000) "Dynamic Optimization for Real-Time Rule-Based Systems using Predicate Dependency," presented at *Sixth IEEE Real Time Technology and Applications Symposium* (RTAS 2000), Washington, DC.

33. [Lee00] Lee, M. Offutt, A. J. and Alexander, R. T. (2000) "Algorithmic Analysis of the Impacts of Changes to Object-Oriented Software," presented at Technology of Object-Oriented Languages and Systems (TOOLS 34'00).

34. [Len95] Lenat, D. B. (1995) "CYC: A Large-Scale Investment in Knowledge Infrastructure," *Communications of the ACM*, vol. 38, pp. 33-38.

35. [Lev00] Levinson R. and Goodwin, A. R. (2000) "Explorations in Scientific Thinking: a Systems Theoretic Approach: Chapter 2," in *Scientific Thinking: a Systems Theoretic Approach.* Santa Cruz, CA, pp. 65.

36. [Luk97] Lukose, D. (1997) "CGKEE: Conceptual Graph Knowledge Engineering Environment," in Conceptual Structures: Fulfilling Peirce's Dream, vol. 1257, Lecture Notes in Artificial Intelligence, D. Lukose, H. S. Delugach, M. Keeler, L. Searle, and J. F. Sowa, Eds.: Springer-Verlag, Berlin, Germany, pp. 598-602.

37. [Luk97] Lukose, D. Delugach, H. S. Keeler, M. Searle, L. and Sowa, J. F. (1997) *Conceptual Structures: Fulfilling Peirce's Dream*, Lecture Notes in Artificial Intelligence, vol. 1257: Springer-Verlag Berlin, Germany.

38. [Mar97] Martin, P. (1997) "The WebKB set of tools: a common scheme for shared WWW Annotations, shared knowledge bases and information retrieval," in Conceptual Structures: Fulfilling Peirce's Dream, vol. 1257, Lecture Notes in Artificial Intelligence, D. Lukose, H. S. Delugach, M. Keeler, L. Searle, and J. F. Sowa, Eds.: Springer-Verlag, Berlin, Germany, pp. 585-588.

39. [Min94] Mineau, G. W. (1994) "Views, Mappings, and Functions: Existential Definitions to the Conceptual Graph Theory," presented at *Second International Conference on Conceptual Structures* (ICCS94), College Park, MD.

40. [Min00] Mineau, G. W. (2000) "The Engineering of a CG-Based System: Fundamental Issues," in *Conceptual Structures: Logical, Linguistic and Computational Issues*, vol. 1867, Lecture Notes in Artificial Intelligence, B. Ganter and G. W. Mineau, Eds. Springer-Verlag, Berlin:, Germany, pp. 140-156.

41. [Pap97] Papazoglou, M. Delis, A. Bouguettaya, A. and Haghjoo, M. (1997) "Class Library Support for Workflow Environments and Applications," *IEEE Transactions On Computers*, vol. 46, pp. 673-686.

42. [Pot97] Potts, C. (1997) "Requirements Models in Context," presented at *3rd IEEE International Symposium on Requirements Engineering (RE'97),* Annapolis, MD.

43. [Pro00] Prost, F. (2000) "A Static Calculus of Dependencies for the□-Cube," presented at 15th Annual IEEE Symposium on Logic in Computer Science (LICS'00), Santa Barbara, CA.

44. [Rum98] Rumbaugh, J. Jacobson, I. and Booch, G. (1998) *The Unified Modeling Language Reference Manual.* Addison-Wesley Reading, MA.

45. [Rya93] Ryan K. and Mathews, B. (1993) "Conceptual Graphs as an Aid to Requirements Reuse," in *Proc. IEEE Symposium on Requirements Engineering.* San Diego, California.

46. [Slea91] Sleator D. D. K. and Temperley, D. (1991) "Parsing English with a Link Grammar," School of Computer Science Carnegie Mellon University, Pittsburg, PA CMU-CS-91-196, http://bobo.link.cs.cmu.edu/link/ http://bobo.link.cs.cmu.edu/link/papers/index.html, October.

47. [Sno98] Snoeck M. and Dedene, G. (1998) "Existence Dependency: The Key to Semantic Integrity Between Structural and Behavioral Aspects of Object Types," *IEEE Transactions on Software Engineering*, vol. 24, pp. 233-251.

48. [Sow84] Sowa, J. F. (1984) *Conceptual Structures: Information Processing in Mind and Machine*, The Systems Programming Series, First ed: Addison-Wesley, Reading, MA.

49. [Sow00a] Sowa, J. F. (2000) "Conceptual Graph Standard," NCITS.T2 Committee on Information Interchange and Interpretation, proposed standard, http://www.bestweb.net/~sowa/cg/cgstandw.htm#Header_15, December 6.

50. [Sow00b] Sowa, J. F. (2000) *Knowledge Representation: Logical, Philosophical, and Computational Foundations*: Brooks/Cole, NY, NY.

51. [Sub00] Subrahamanian, V. S. Bonatti, P. Dix, J. Eiter, T. and Ozcan, F. (2000) *Heterogeneous Agent Systems*, 1st ed. MIT Press, Cambridge, Mass.

52. [Tha98] Thalheim, B. (1998) *Entity-Relationship Modeling: Foundations of Database Technology.* Springer-Verlag, New York, NY.

53. [Tha98] Thanitsukkarn T. and Finkelstein, A. (1998) "A Conceptual Graph Approach to Support Multiperspective Development Environment," presented at *11th Workshop on Knowledge Acquisition*, Banff, Alberta, Canada.

54 [Web80] (1980) Websters New Collegiate Dictionary, G and C Merriam Company, Springfield, Massachusetts.

55. [Yu96] Yu, E. S. K. Mylopoulos, J. and Lespérance, Y. (1996) "AI Models for Business Process Reengineering," *IEEE Expert*, vol. 11, pp. 16-23.

APPENDIX A CONCEPTUAL GRAPHS

A 1. Purpose

Conceptual graphs are intended as a general knowledge representation to help solve problems in natural language understanding, human decision-making, behavioral modeling, and many other important areas of interest. CG's are typically used by representing knowledge in some domain of interest that is relevant to the problem being addressed. The knowledge in CG's may be translated from some other representation (e.g., UML), constructed through some knowledge acquisition process, or it may be drawn directly through a conceptual graph editor and incorporated into a system.

A 2. History

Conceptual graphs are the culmination of several directions in cognitive science, semantic networks, and formal logic. These topics were synthesized into conceptual graphs by John Sowa in his pivotal 1984 book (Sowa 1984). They resemble C.S. Peirce's existential graphs from the late 1800's. One of the original motivations for developing conceptual graphs came from the database modeling community, where Chen's entity-relationship diagrams [Che76] had been in wide use. CG's overcome some of the limitations of ER diagrams (see Strengths below).

Since Sowa's original work, there have been seven international workshops followed (starting in 1993) nine international conferences with a major focus on conceptual graphs. Work in formal concept analysis and KIF has been closely integrated with conceptual graphs for the past several years. The conference proceedings have been published regularly by Springer-Verlag in their series of Lecture Notes in Artificial Intelligence.

A 3. Basic Introduction

A conceptual graph system consists of a knowledge base and an implemented set of operations to manipulate the graphs comprising that knowledge. A knowledge base itself consists of a set of concept types, a set of relation types, a set of graphs representing what is known about the domain of interest and a set of individuals in that domain.

A 3.1 Concepts, Relations and Types

The notion of a "concept" is the basis for modeling knowledge in conceptual graphs. A concept can be any distinguishable idea, including a process, an attribute or an individual. Each concept has a particular type. In its graphical form, a concept is shown as a type-labeled rectangle, as in Figure 29.

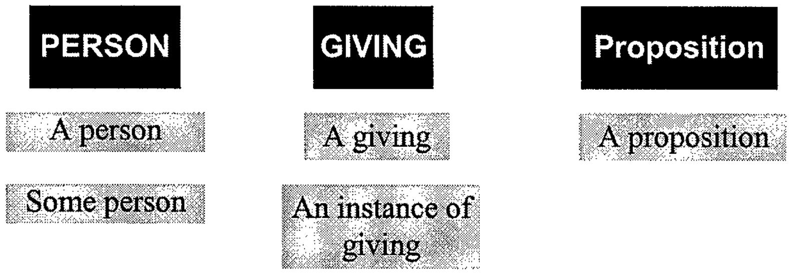


Figure 29. Sample Concepts.

Other components of a conceptual graph will be introduced shortly. Graphs are drawn on a two-dimensional surface called a **sheet of assertion**, which represents all that is "known" about the domain of interest. A conceptual graph therefore constitutes an assertion about the domain. In particular, a concept drawn on a sheet of assertion represents an instance of that concept.. In terms of first order predicate calculus, the graphs represent an existential assertion; *e.g.*:

There exists an individual person

There exists an instance of giving

Concepts can be linked to one another by **relations**, which are each defined with a particular semantics and constraints on the concepts they may link together. A relation links two or more concepts (or context,s, see below). A relation is shown as a circle or oval, as in Figure 30.

Concept and relation types form the set of allowable labels on concepts and relations.

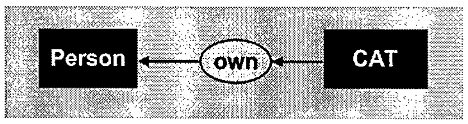


Figure 30. Example Relation.

The arrows of a relation may be arbitrarily defined, but a conceptual graph system must maintain a consistent interpretation throughout. There is a linguistic convention applied to the direction of the arrows in a relation. If arrows on relation R go from A to B, then we want to say:

The R of A is B

For example, in Figure 30, we would interpret the relation to say:

The owner of a CAT is a Person.

In order to relate type labels to one another, a type hierarchy of concepts types is maintained in a conceptual graph system. The type hierarchy captures the "is-a" or "is-a-kind-of" relationship between types, much like inheritance in an object-oriented sense. It is therefore different from a relation, since it shows relationships between types themselves and not instances of those types. One type can be a super-type to zero or more sub-types, so that multiple inheritance is allowed. Relation labels have their own separate hierarchy of type labels. For example, a simple hierarchy is shown in Figure 31. At the top of the hierarchy is the **universal type T** which represent "all things" or the type "Thing" in Cyc. At the bottom of the hierarchy (to complete the lattice) is the type _|_ representing "no thing" called the **absurd** type. (For simplicy, we generall show a CG type lattice as a hierarchy, with the absurd type implied below all "leaf" types in the hierarchy.)

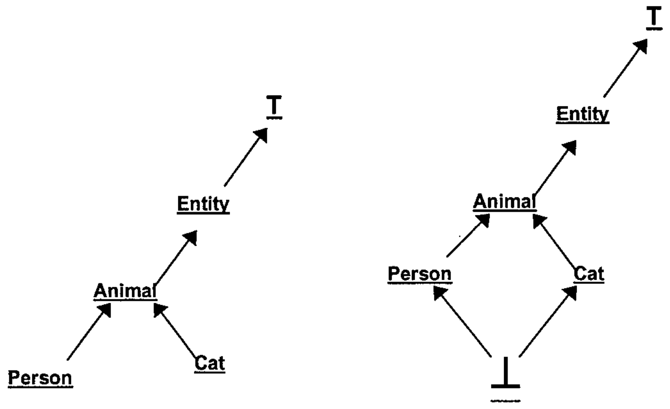


Figure 31. Hierarchy (Lattice) Of Type Labels.

Super- and sub-type relationships can also be shown textually as

Entity < T.

Animal < Entity.

Person < Animal.

Cat < Animal.

Both of these representations mean the same thing

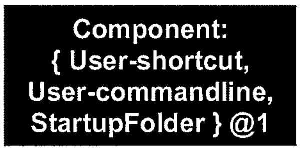
Entity is a kind of universal type.

Animal is a kind of entity, etc.

A 3.2 Referent

In addition to its type, a concept can have several distinguishing constraints included. These constraints are collectively called its **referent**. Some examples of a referent are shown in Figure 32. There are additional kinds of referents to co-identity with other concepts (not shown here). These constraints, at a low level, could always be shown as additional concepts themselves (e.g., a concept for each individual linked to each of its possible types), but using a referent makes the notation more compact and easier to read.

Person	Some person
Person: Dan	Person Dan
Person: #95575	Person numbered 95575



One of set of components named User-shortcut, User-commandline and StartupFolder

Figure 32. Referent Examples.

A 3.3 Canonical Graph

In order to establish a consistent meaning across multiple graphs and knowledge bases, a conceptual graph system must have a set of **canonical graphs** defined for it.. A canonical graph constraints concept types to a supertype or any of its subtypes, and also defines the relations that are allowed between those concepts. By defining a syntax for specific concepts and relations, a canonical graph provides semantic constraints for what a CG system can say. This prevents nonsense or gibberish from being included as graphs. Canonical graphs therefore serve a similar purpose as the grammar rules in a formal textual language. Figure 33

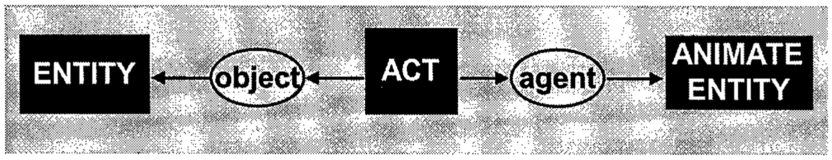


Figure 33. A Canonical Graph.

"The object of an act is an entity and the agent of an act is an animate entity."

Conceptual graphs can thus specify semantic constraints via canonical graphs and enforce semantic consistency by comparing graphs in the knowledge base with those canonical graphs. A "badly-formed" graph can be easily detected.

We will now illustrate how semantic inconsistency is detected. As an example, consider a CG system where the graph in Figure 33(b) is specified to be a canonical graph.

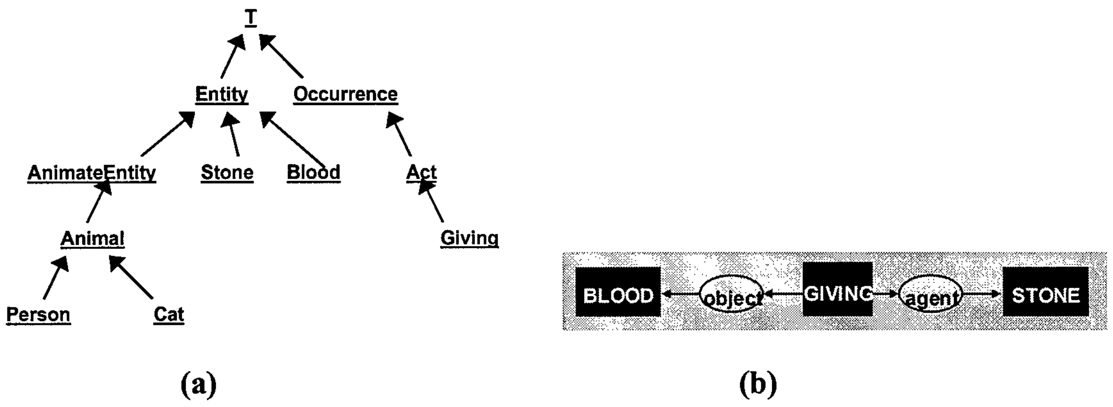


Figure 34. Semantic consistency example graphs.

Consider the hierarchy and graph in Figure 33. The graph in Figure 33(b) represents the knowledge that "a stone is giving blood". To compare this with the canonical graph in Figure 33, a CG system would detect that GIVING is a subtype of ACT, whose canonical graph shows that it must be linked via relation agent to an ANIMATEENTITY (or one of its subtypes) and that it must be linked via relation object to an ENTITY (or one of its subtypes). In the graph of Figure 34(b), the type BLOOD is a subtype of ENTITY and thus meets the semantics constraints, but the type STONE is not a subtype of ANIMATEENTITY and therefore violates the constraints implied by the canonical graph. A CG system could also provide feedback to the system providing the knowledge as a potential "error" in the originating system.

A 3.4 Contexts and Rules

One powerful feature of conceptual graphs is the ability to capture a context – a collection of conceptual structures that themselves form a new concept. This "factoring" ability allows conceptual graphs to be arbitrarily nested, increasing understandability as well as providing new mechanisms for inferencing and constraints matching.

For example, the graph in Figure 35 shows the rule "If it is raining, then the ground is wet." In logic terms, the rule actually states that "it is not the case that it is both raining and the ground is not wet.." (Note that this is a dependency of type Unknown Causality, since there is no mention of process or temporal knowledge; merely a correlation is being shown.) The default type for a context is the type Proposition.

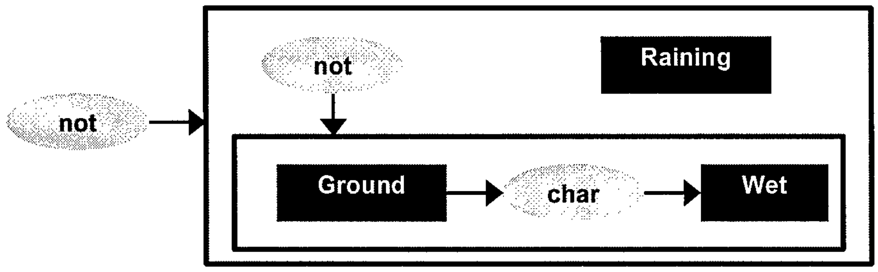


Figure 35. A Conceptual Graph Rule Using Contexts.

Contexts allow entire graphs to be manipulated as a single concept., facilitating treatment of specifications at varying levels of detail; for some purposes a context may be "collapsed" or "expanded" as needed, either by the system for inference purposes or by a human user for understanding.

A 3.5 Operations

There are standard first order logic inferencing mechanisms, which (based on C.S. Peirce's existential graphs) consist of graph rewriting rules, as described in [Sow84]. These rules permit transformations of conceptual graphs according to accepted principles of logic. As a simple example, suppose we have the rule in Figure 35 asserted, along with the single concept [Raining]. Conceptual graph transformation rules allow us to imply: "the ground is wet." as in Figure 36.

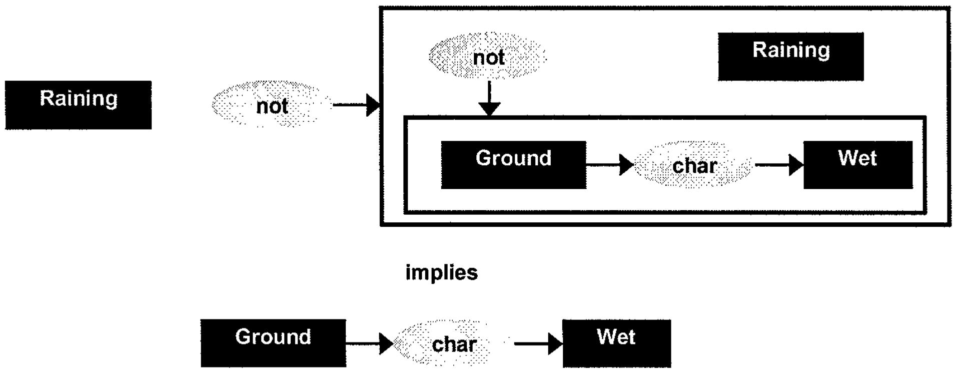


Figure 36. Implication In Conceptual Graphs.

The complete set of conceptual graph transformation rules is beyond the scope of the overview; see [Sow84] and [Sow00b] for details.

A 3.5.1 Join

The join operator combines conceptual graphs so that more complete descriptions can be obtained. It is also how we establish indirect relationships from direct ones. The example given here shows how

A 3.5.2Projection

The projection operator can be very useful in accessing conceptual graph knowledge bases, particularly when complete graphs are not desirable or necessary to analyze a problem. The projection operator identifies subgraphs that match a given layout, obeying rules about subtypes and individuals. For example, suppose we want to identify all dependencies involving the dependency feature Need. Suppose we have the graph in Figure 37. It represents a generic dependency with only a Need feature.

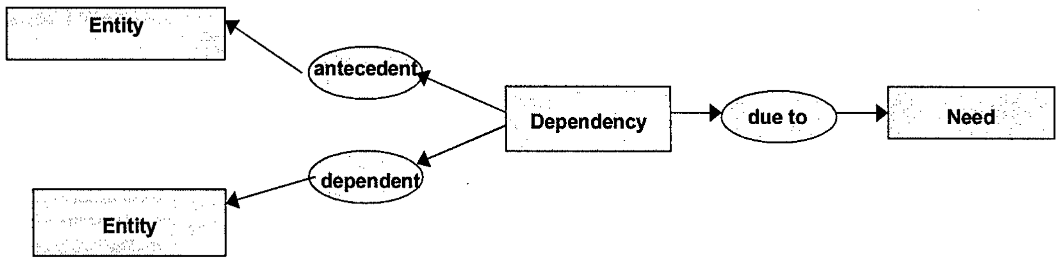


Figure 37. Graph For Projection.

Projecting this graph onto the graph in Figure 23 is the logical equivalent of analyzing Figure 23 and identifying all the dependencies that are due to need, while effectively ignoring everything else in the graph, whether related to dependency or not. The results of the projection would be a series of subgraphs, such as those shown in Figure 38. The operation of projection thus has the effect of trimming a graph down to subgraphs chosen by picking a "template" graph.

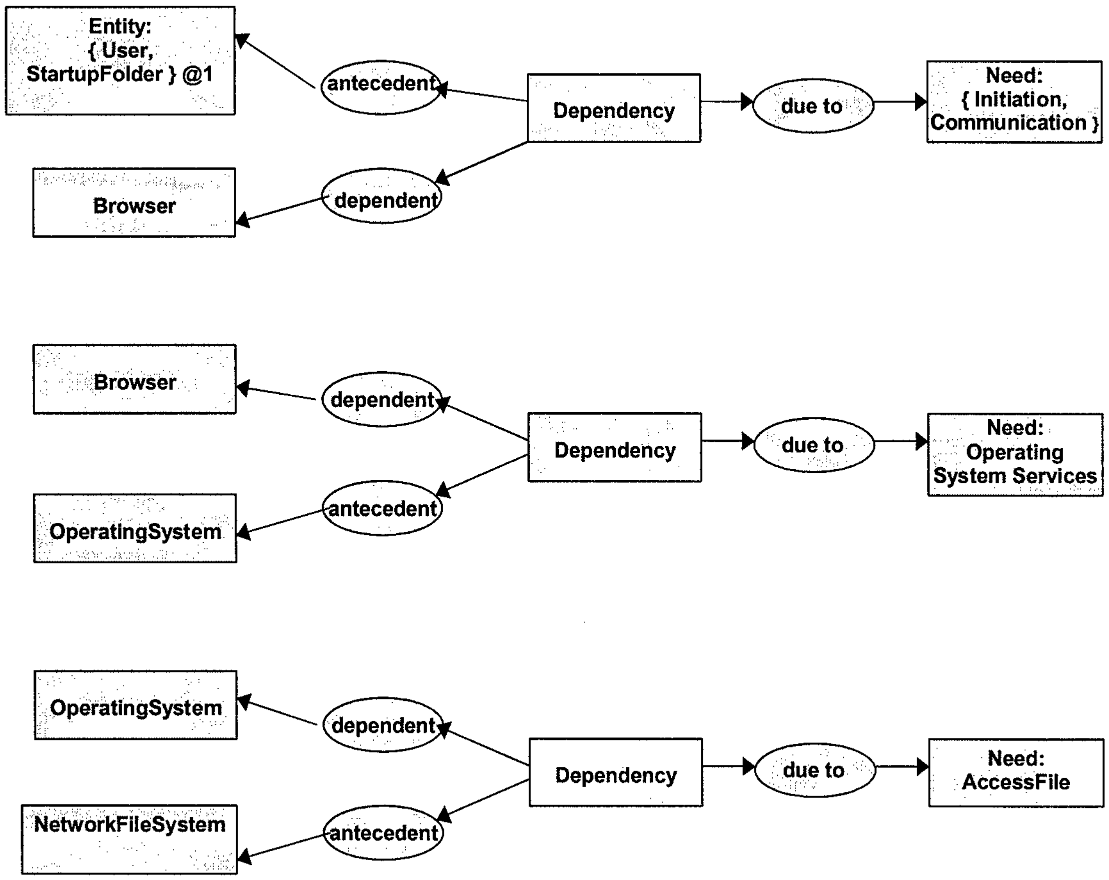


Figure 38. Subgraphs Resulting From Projection.

A 3.6 Uses

Conceptual graphs have been used in a variety of knowledge modeling and enterprise modeling environments. Among the more interesting applications of conceptual graphs are in natural language translation and paraphrasing [Ang00], database semantics and interoperability [Car94] [Del92], document retrieval systems [Gen97], modeling and simulation of human behaviors [Bos97] [Bal00], software requirements modeling [Del91] [Del92] [Rya93] [Tha98], computer-supported cooperative work [deM97]

A number of researchers have developed research-quality tools that support conceptual graphs and their operations at various levels. One of the earliest was developed at Deakin University in Australia [Gar97]. A CG tool that supports natural language processing and basic graph drawing has been developed in Sofia, Bulgaria [Ang97]. A text-based tool for supporting CG operations was constructed by Lukose [Luk97]. Graph editing and some operations are supported by tools we have developed, e.g., CharGer [Del00a] and CORE, [Bev00] as well as tools developed in France [Che97]. A Web-based knowledge-modeling tool called WebKB has also been developed [Mar97]. A general discussion of CG tool requirements can be found in [Min00].

A 4. Evaluation

A 4.1 Strengths

Powerful representation. Conceptual graphs are able to represent concepts, relations, nested concepts, activities, functional relationships and assertion/retraction of graphs, making them a complete system for knowledge modeling and analysis. Conceptual graph systems have wide utility, and represent a mature technology, having been extensively modified and augmented since 1984. CGs possess more power than ER diagrams, in that multiplicity can be represented beyond the capabilities of ER diagrams. For a full discussion, see [Sow00b]

Scalable. Much work has been done to establish scalability for conceptual graphs. Compact and efficient representations have been explored for years, such as in [Ell95] and [Min00]. Large conceptual graph systems have been built and tested.

A 4.2 Weaknesses

Knowledge Acquisition. Conceptual graphs' main weakness is in the area of knowledge acquisition. Because a conceptual graph system has a number of underlying constraints, such as type definitions, relation definitions, canonical graphs, actor definitions, etc., it is sometimes difficult for graphs to be drawn "by hand" except by a knowledge engineer who is well-versed in conceptual graphs and all of the underlying definitions of a particular system. We anticipate that other established techniques (supported by the conceptual graph system) will be used for acquiring knowledge from system develops, such as repertory grids and formal concept analysis [Del00b].

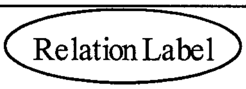
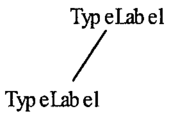
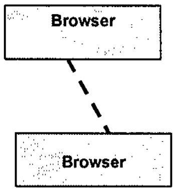
Lack of guidance for the system builder. Another weakness is the lack of an accessible reference book to describe how conceptual graphs systems are constructed. We are working directly on this problem too, with a book in process [Del02] that will assist system builders in knowing what issues to address and giving guidelines on how to implement various CG features for their purposes.

A 4.3 Potential

With the ability to operate directly on the knowledge aspects of dependencies, conceptual graphs not only possess the capability to analyze and detect dependency problems, they can also be incorporated into a general analysis system whereby many questions (not just dependency) can be addressed. For example, testing and deployment issues may also be addressed by a conceptual graph system. Once a detailed model of a system is obtained in conceptual graphs, many issues can be addressed "for free" since the knowledge model is general and comprehensive.

Definitions can be established for transitivity in certain dependencies, equivalence or non-equivalence in other dependencies, etc. so that different users can obtain different views of the system's characteristics. Projection can support trimming a large graph into a smaller, more understandable one. Join can establish relationships between components that might be overlooked in a more conventional analysis system, particularly if the relationships or interactions are subtle. If we bring the full power of a CG system to bear on this set of issues, a number of new benefits accrue.

LIST OF SYMBOLS

$\boxed{\text{TypeLabel: Referent}}$	Concept with type label and referent (in conceptual graphs)
	Relation with relation label (in conceptual graphs)
	Conceptual type hierarchy inheritance (in conceptual graphs)
	Line of identity, or Co-referent link In conceptual graphs
$\in$	An element of, in set theory
$>$	Greater than
$\binom{N}{2}$	Number of combinations of N things taken 2 at a time
$\top$	The universal type in conceptual graphs (see Appendix)
$\perp$	The absurd type in conceptual graphs (see Appendix)

GLOSSARY

Absurd Type	The type at the "bottom" of the type lattice in conceptual graphs, representing the type that is "no thing" (see Universal Type)
Antecedent	In a dependency relationship, the thing on which one or more other things depends
Canonical Graph	A graph in a CG system that defines the syntax form of typed-concepts and relations thus constraining the semantics of what can be said in a graph.
Concept	A distinguishable thing, such as an entity, process, belief, action, etc. in conceptual graphs
Conceptual graph	A collection of concepts, relations, and/or actors linked together to form a statement of knowledge, possibly nested, and generally considered to be part of a larger knowledge base to represent a domain of interest.
Cyc	A large knowledge base and inferencing system developed by Cycorp with the intent to capture the knowledge of a human.
Dependency	Any relationship between two or more things such that a change in one or more of the things can potentially cause a change in one or more of the other things.
Dependent	In a dependency relationship, something which depends on an antecedent
Email	Electronic Mail, frequently transmitted over Ethernet
Ethernet	LAN Hardware and Software Protocol providing computer-to-computer communications over limited distances.
Excel	Microsoft Spreadsheet Program
Lattice	A mathematical structure whereby elements stand in partial order to each other, such that there is a single top-most (i.e., highest ordered) element and a single bottom-most (i.e., least-ordered) element.
Linux	Open Source Version of the Unix Operating System
MS Explorer	Microsoft internet browser. A competitor to Netscape.
Netscape	An internet browser

Outlook	An email client produced by the Microsoft Corporation
Outlook Express	An email client produced by the Microsoft Corporation
Requirements	A phase or aspect of software and system development during which needs are analyzed, basic system architecture is studied and the overall goals of the software or system are determined.
Referent	Constraining or identifying information about a concept in CG's.
Relation	A conceptual graph relationship between two or more concepts or contexts
Sheet of Assertion	A logical knowledge space where conceptual graphs (and hence the knowledge they represent) are asserted.
Statement of Work (SOW)	Our governing contract materials, in particular the list of tasks to be performed by this research effort.
Transitive dependency	Any dimension or attribute of dependency (e.g., access dependency) that obeys the property of transitivity; i.e., where elements in a transitivity chain are, in fact, dependent on their predecessors in that dimension.
Transitivity	The property of unidirectional relations (*e.g.*, dependencies) whereby a "chain" of similarly related things can be analyzed in sequence.
Transitivity Chain	A sequence of two or more elements, where each preceding element is ordered with respect to its succeeding element, and so on down the sequence.
Universal Type	The type at the "top" of the type lattice in conceptual graphs, representing the type that is "all things" (see Absurd Type)

INDEX / CONCORDANCE

Dependency Analysis Using Conceptual Graphs

Lisa Cox, Dr. Harry S. Delugach

Computer Science Dept.
University of Alabama Huntsville
Huntsville, AL 35899
lcox@cs.uah.edu, delugach@cs.uah.edu

Dr. David Skipper

Bevilacqua Research Corp.
Huntsville, AL 35815
DavidS@brc2.com

Abstract. Analysis of dependencies between entities is an important part of modeling. Whether the modeling domain is at the enterprise level or at the system or software component level, characterization, representation, and analysis of these dependencies is essential to correctly modeling the domain. For example, it is important to identify and characterize dependencies between both system and software components when trying to determine the extent of and impact of a breach in computer system security or of a malfunction in a component. Analysis of such dependencies is also greatly beneficial in both the requirements and maintenance phases of software engineering. What is needed is a formal characterization of the concept of dependency along with a more formal and unified approach to dependency analysis . This paper introduces the notion of dependency at a general level. In the present literature, an actual definition and characterization of a dependency is usually avoided, and it is difficult to separate the discussion of the dependency from the particular domain of interest. Most of the literature available implies that it is simply "understood" that a dependency can be represented by a directed arc on a graph where the dependent components are the nodes of the graph. Much work in the current literature addresses dependencies in widely varying ways. This paper attempts to formalize both the definition and characterization of a dependency in a unified approach, and then illustrates how dependencies themselves and the effect of those dependencies upon a system can be efficiently modeled using Conceptual Graphs.

1.0 Introduction

In modeling, it is usually important to identify, characterize, and understand the impact of the dependencies that exist between the entities in the model. This is vital at all levels of modeling and in all domains. The concepts and approach presented in this paper are applicable at all levels and in each domain. We start by considering that there are many notations in use for the specification of and analysis of models. While some of these notations allow explicit specification of dependencies, some include dependency only by implication. For example, UML represents a simple dependency as a dotted arrow between components. [1]

In the software engineering domain, OSD (Open Software Description) is presently being pushed by Microsoft as a standard for describing and packaging software [5]. While OSD and the literature surrounding it are in agreement that dependency representation is important, OSD simply represents dependencies by supplying a list of other components that are required to be present before a particular software component can be installed on a system. [6]

Work has also been done in the natural language processing area dealing with dependency analysis between words of a sentence, and specific linguistic dependency types have been identified, such as the dependency between a noun and a determiner or the dependency between noun and verb. [16], [3] However, these dependencies once again are limited to the particular domain in question (i.e., linguistic dependency) and explicit definitions of an abstract dependency are not considered.

Much of the present literature takes the definition of dependency for granted and where definitions are occasionally given, they vary widely. Some sources maintain that dependencies are simply first-order logic formulae, or in database terminology, constraints [18], [4], [5]. Others insist that higher-order logic is required to express dependencies. [14] Some take a probabilistic approach and express dependencies as conditional probabilities between specified variables or look solely at dependencies from a statistical viewpoint. [10], [17], [2]. Some sources take the approach that a dependency is best modeled by the client/server relationship, and then develop the definition of dependency in client/server terms [15], [19], [8], while others specify types of dependency such as structural and functional dependencies [8] or data and value dependencies. [13].

Keller, Blumenthal, and Kar in [8] attempt a more in-depth characterization of dependencies and define six different "dimensions" of dependency, and Prost in [14] also takes a "type-based" approach to dependency analysis. However, some of the dimensions given in [8] for analyzing dependencies are actually attributes of the computer system under analysis. Once again, there is no clear delineation between the dependency itself, and the domain in which the dependency exists.

Mineau in [12] discusses the addition of functions to and the treatment of functional dependencies in Conceptual Graphs, but even Mineau does not address the explicit definition of a dependency.

This paper presents an approach for formally defining and characterizing dependencies using Conceptual Graphs. It is our contention that our approach to the definition of dependency and the use of Conceptual Graphs as a dependency language allows for a much more coherent and complete description of dependencies at the general level and explicitly delineates the characteristics of the dependency from any domain limitations. We also expect the use of Conceptual Graphs to allow more powerful analysis of the dependencies of a given system.

2.0 Definitions

Our perspective comes from the Realist's view as defined by Hayes in [7]. We assume "a set can be a set of anything" and that "the universe can be physical or abstract or any mixture" in order to make our universe as general as possible.[7] Based upon this perspective, we then refer to an entity as anything that can be a member of such a set, and therefore can be anything we want to model. This can be an object, a concept, an organization, or any other thing to be modeled. We also make the assumption that the entities are not static. The entities can change. At this point, we simply assume the existence of something called change that happens to entities, but we deliberately do not yet attempt to define change in order that it, too, may be allowed to be as general as possible. We understand that an entity may change for at least several and possibly many reasons. The entity may have change as part of its very nature (for example, try to model a 2-year-old child without allowing for change). The entity may also be influenced to change by something outside itself. This latter type of change is of specific interest to us and it is upon this that we base our understanding of dependency. From this understanding, we assume that there are cases where the "something outside itself" possibly or potentially influences the entity to change.

We ask the reader to accept our general definitions for entity, change, and potential for change in the interest of concentrating upon dependency. We also assume the existence of a relation R between some number of entities, expressed by R(A, B, C, D, . . .) where it can be said that the R relationship exists between the entities A, B, C, D, etc.

In the general case, we define a dependency as such a relation, D, between some number of entities wherein a change to one of the entities implies a potential change to the others. We can therefore express such a general dependency as D(A, B, C, D, . . .) where $D \in R$. This general form of a dependency is shown in Figure 1. In order to emphasize the complexity of this most general type of dependency (which may exist between many entities), we refer to it as symbiosis.

As an example of this most general type of dependency, or symbiosis, we can consider the relationship between the departments within a corporation. It is easy to see that the engineering, accounting, contracts, marketing, and facilities departments are dependent upon each other. However, it is not at all easy to specify and quantify the extent of such a dependency.

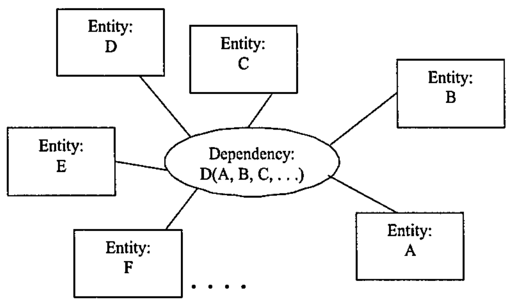


Fig. 1. Graphical representation of most general form of a dependency

As a first step in our analysis, we focus upon a much simpler type of dependency, the case of a dependency between only two entities, D(A, B). In the case where A depends upon B and B depends upon A, this dependency can be seen as a bi-directional relationship. We call this bi-directional dependency an interdependency. Given such an interdependency between two entities, we can now separate the dependency D(A,B) into at least two one-way, or unidirectional dependencies d1(A, B) and d2(B, A). We can be sure that this is always the case, because we have included "independent" in our type hierarchy for dependencies (refer to Figure 4).

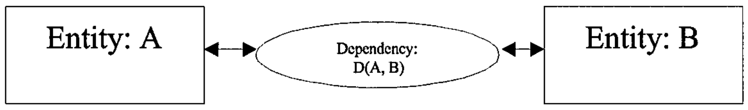


Fig. 2. Bi-directional dependency, or interdependency, between two entities

In the simplest case of a dependency, a unidirectional dependency between two entities, d(A, B), we can say that A depends upon B. If A depends upon B, then a change in B implies a potential or possible change in A. As in Keller, et. al.[12], we refer to A as the dependent and B as the antecedent. This definition of the simplest form of a dependency is very like the definition of dependency given in [1] and is depicted in Figure 3.

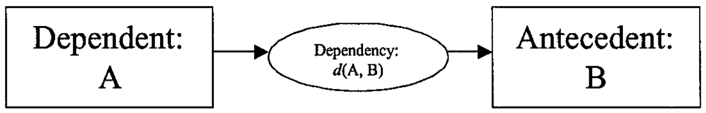


Fig. 3. Graphical representation of the simplest dependency

Again, it is important to note that this definition of the simplest case of dependency expresses a one-way direction for the dependency. As Briand, Wust, and Lounis [2] point out, it is not only possible, but common that a bi-directional dependency exists; and, given the definition of the most general form of dependency above, it is also conceivable to have such an interdependency demonstrated between N entities where N>2.

Our initial work is based upon the decomposition of complex dependencies into unidirectional, binary relations. The complex dependency can be broken into some number of unidirectional dependencies. As described above, it is easy to see that in the case of an interdependency between two entities, the bi-directional dependency can be described using at least two one-way dependencies between the two entities. We expect that in a case of symbiosis among N entities, the symbiosis can be represented by at least

$$2({}^{N})$$

unidirectional dependencies. We use the term "at least" here because there may be multiple types of dependency existing between any two entities. For example, both an intermittent, time-based dependency and a static structural dependency may be involved in the interdependency. Even if the dependency is of a single type, such as a functional dependency, it could include several different and specific "needs" of the entities. In that case, a separate unidirectional dependency could be defined for each specific need. Our continuing research will include a more in-depth investigation of this expectation.

3.0 Describing a dependency

Now that we have defined both a general dependency and the most simple dependency, we need to discover the characteristics that are inherent in all dependencies and we need to investigate the types of dependency that are possible. Our research is focusing on the very ambitious attempt to produce what might be called an ontology of dependencies. This includes both the identification of a set of attributes which apply to every dependency and the development of a general dependency type hierarchy based upon those attributes.

3.1 Attributes which describe a dependency

Keller, et. al.[8] is the only source in which we have found an attempt at the classification of dependencies based upon such attributes. Keller, et. al.[8] lists six attributes of dependency which are represented as orthogonal axes in a six-dimensional dependency space wherein each dependency can be graphed. Our initial set of attributes, which are applicable to all dependencies, includes two attributes from [8], criticality and strength. However we believe that the other four attributes cited by [8], rather than being associated with the dependency, would be more properly represented as attributes associated with the system components (the entities A and B) or with the system, itself. For example, the "component type" cited by [8], is not an attribute of a dependency as much as it is an attribute of the entity, A, being modeled, and the attribute "dependency formalization" is actually dependent upon the particular system in question.

To the two attributes we have taken from [8], we have added the attributes of impact, sensitivity, stability, and need as important to all dependencies. Keller et. al. [8] also addresses the issue of "time", although it is not included in the six-dimensional dependency space. This is very like the attribute we have named stability. The following represent our current definitions of our initial set of attributes:

> **Sensitivity** (or fragility) – how vulnerable to compromise or failure is this dependency? Possible values for this attribute are Fragile, Moderate, and Robust.
>
> **Stability** (like "time" in [8]) – a measure of the continuity of the dependency's vulnerability to compromise or failure (sensitivity) over time. One way of looking at stability is to ask the question: "When is the dependency fragile?" Possible values for this attribute are Extremely Stable, Infrequent, Periodic, Certain Defined Times only, etc.
>
> **Need** – what "need" of entity A is fulfilled by entity B? This can be expressed as a list of particular capabilities upon which this dependency is based. Possible values for this attribute include Authorization, Resources Provided, Testing, or at lower levels could include Text Editing, Computation, Network Access, File Save/Retrieval, etc.
>
> **Importance** (or criticality) – what is the weight of this dependency as a determinant of entity A's success, or how critical is this dependency to the goals and overall function of entity A? Possible values for this attribute are: Not Applicable, High, Medium, and Low.
>
> **Strength** – a measure of the frequency of the need or the importance of this dependency, from entity A's viewpoint. How often or how much does entity A rely upon this dependency in any particular time period? One way of looking at Strength is to ask the question: "How often does this dependency's importance or need come into play?" Possible values of this attribute are Daily, Hourly, Yearly, etc. or a numeric value representing how often the dependency is an issue during a particular time period.
>
> **Impact** – in what way is the entity's function affected by compromise or failure at this particular dependency? Possible values for this attribute are: None, Mission Compromised, Information Unreliable, Performance Degraded, Corruption/Loss of Information/Communication.

This represents our initial attempt to identify the set of attributes which are applicable to all types of dependencies. Using this initial set of attributes, we are able to determine an initial version of a hierarchy of dependency types. We expect that if it were possible to identify a complete set of such attributes, that we should then be able to identify all possible dependency types in our hierarchy.

3.2 Dependency type hierarchy.

Once a complete set of dependency attributes is identified, it will then be possible to establish a type hierarchy, resembling a lattice, based upon those attributes and their values. Using this hierarchy, specific types of dependency are characterized and related to each other, and dependency types can be chosen to be applicable to particular domains. Eventually, it should be possible to fully populate the dependency type hierarchy based upon the attributes identified. Figure 4 contains a portion of the dependency type hierarchy identified so far. From the types shown in this structure, it is now possible to analyze the dependencies discussed by each of our sources and indicate where in the structure their particular approach to dependency lies.

Several of our sources assume no more detail about a dependency than that it is a directed arc between two entities. [3], [5], and [9] are examples.

Mineau [12] goes into specifics about functional dependencies. In Lukose and Mineau[11], data dependencies are explicitly referenced, and the messages passed between actors can also be seen as data dependencies. Lukose's "control marks" and "control maps" also represent dependencies which are reflected in our hierarchy as the control dependency type. These dependencies appear to lie near the bottom of our structure as functional, data, and control dependencies respectively.

Lukose and Mineau [11] also discuss co-reference links between concepts. A co-reference link indicates that two concepts exist which describe the same entity. We have allowed that two or more representations for the same entity may be required and have included a co-reference dependency in the structure.

Briand et. al. [2] refers to data and control dependencies in particular metrics analyzed, but also introduces the dependency between object classes based upon inheritance. That type of dependency lies in the "requires" section of the structure as can be seen in Figure 4.

4.0 Using Conceptual Graphs as a dependency language.

Given the definition of dependency above, it is now straightforward to map dependencies into conceptual graphs. First, the definition of the simplest dependency is encoded in Conceptual Graph terms. Figure 3, depicting such a dependency is already in Conceptual Graph form. From there, the graph may be relationally expanded to include the definition of the dependency using its attributes. This conceptual graph is shown in Figure 5.

The relation, "dependency" has now been expanded into a graph defining a concept of "Dependency" which is related to the previous concepts of

"Dependent" and "Antecedent" and which is also now associated with attributes characterizing the most general dependency.

Note that the conceptual graph representation allows us to easily represent the most general case and also to expand the general graph in order to represent more specific information about the dependency as it becomes available, i.e. the Conceptual Graph representation facilitates modeling at multiple levels of detail simultaneously. This addresses one of the most difficult problems in modeling, the efficient representation of and processing of entities modeled at multiple levels of fidelity. Using Conceptual Graphs, the scalability problem becomes much less difficult and in some cases is solved altogether.

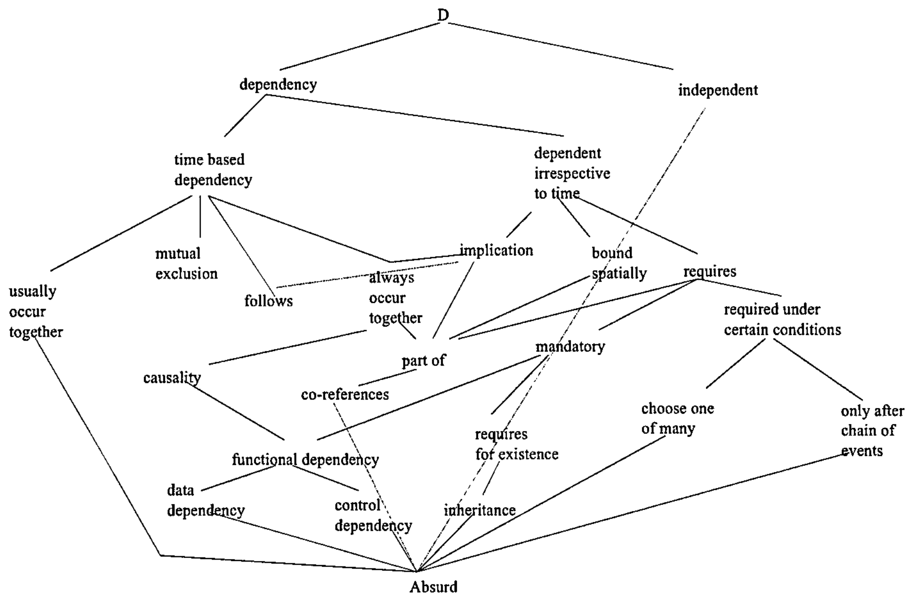


Fig. 4. Dependency type hierarchy

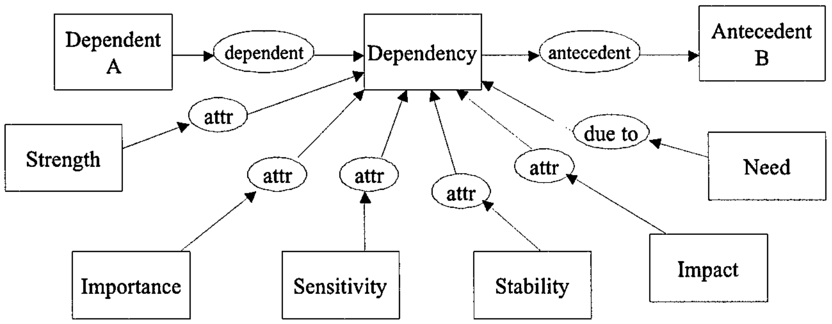


Fig. 5. Relationally expanded dependency graph

For applications such as those in [3], that need no more knowledge of the dependency than the direction of the relationship (or the directed arc), the graph shown in Figure 3 need not be expanded at all. Also note that if the parts of speech used in [3] are defined as subtypes of "Dependent" and "Antecedent", then restriction of the graph shown in Figure 3 allows the expression of all the dependencies used by that source.

As noted earlier, [8] focused upon dependencies between hardware and software system components in a distributed system. All six dependency attributes cited in [8] are vital to analysis of that domain. While our dependency attributes specifically include two attributes from [8], the issues surrounding the other four must also be addressed. In order to address the others, we first need to restrict the dependent and antecedent to a subtype of "computer system component." This representation allows the information pertaining to the system components to be put into attributes associated with those concepts in order to leave the definition of the dependency concept uncluttered. A possible definition of "computer system component" which will include the information required by [8] is shown in Figure 6.

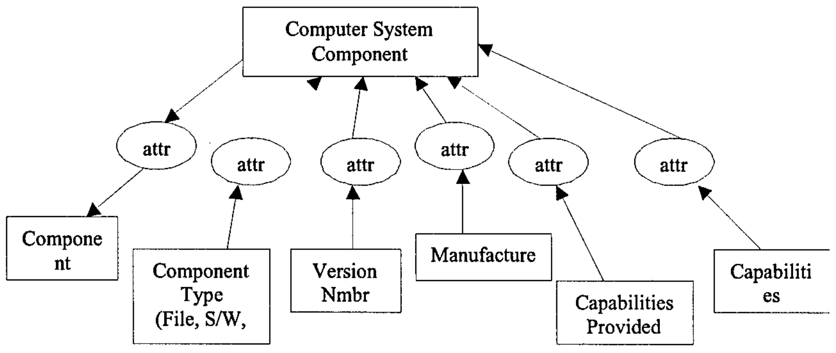


Fig. 6. Computer system component definition

In this way, the two requirements for "component type" and "component activity"[8] (which we have named "capabilities provided") are represented as attributes of the components and thereby influence the analysis, but are separated from the dependency itself.

The dimension of "locality"[8] is more difficult to deal with. But if we introduce the idea of dependency chains, as indeed were introduced in [8], then a dependency can be defined between entities A and E, d1(A, E) where the set of dependencies, {d2(A, B), d3(B, C), d4(C, E)} form such a dependency chain. This introduces a limited transitivity of dependencies: given the possibility that d2, d3, and d4 can be of different dependency types, it is very difficult to draw conclusions about the nature of such transitivity without examining the specific definitions of the dependency types. However, if d2, d3, and d4 are identified as dependency types that are indeed transitive, then the attribute of "locality"[8] can then be implemented by counting the "hops" on the fully expanded dependency chain and including a weighting factor or "importance" such that a dependency "hop" between a software component and a hardware component is more significant than one between two software components.

We also have not addressed the last dimension given [8]. "[D]ependency formalization" does not appear in our attribute list. Keller et. al. define that particular dimension as "a metric [signifying] how expensive and/or difficult [it is] to acquire and identify this dependency," particularly relating to the "degree it can be determined automatically." Although we understand why this particular "dimension" is important given the domain of focus, we again think it is better to separate this from the attributes of the general dependency. In some systems a dependency may be extremely simple to "determine automatically" if UML descriptions of system components are available, while an identical dependency may be extremely difficult to identify "automatically" in a legacy system which has little supporting documentation.

5.0 Our approach applied to two examples

To illustrate the application of our approach, we now present two examples. In the first, we consider the analysis of a particular dependency in a computer system from the information security perspective. A complex dependency exists between network browser (Brows), e-mail package (EMail), word-processing package (WP), and the hardware/software that provides network access (Net). That dependency may be broken into at least twelve unidirectional dependencies as follows.

d1(Brows, EMail)	d2(Brows, WP)	d3(Brows, Net)
d4(EMail, Brows)	d5(EMail, WP)	d6(EMail, Net)
d7(WP, Brows)	d8(WP, EMail)	d9(WP, Net)
d10(Net, Brows)	d11(Net, EMail)	d12(Net, WP)

In systems where the network browser and the e-mail package are unrelated, d1 and d4 will be extremely weak dependencies, to the point where they are categorized as independent. In other systems, where the same software package is used to provide both these services, the dependencies will be much stronger. In the case where the word processing package, WP, depends upon the network browser, Brows, to download new fonts, the dependency d9(WP, Net) can be replaced by the dependency chain: { d7(WP, Brows), d3(Brows, Net) }.

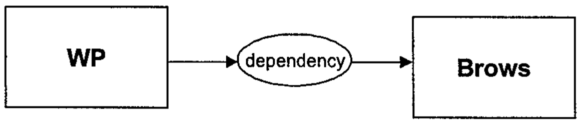


Fig. 7. Dependency Example

In this example, the attributes of each dependency could be given values where data is available, but left without a value if the attribute does not involve itself in the analysis.

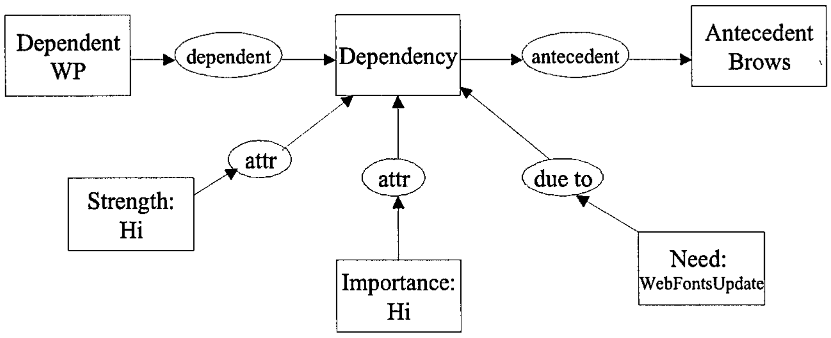


Fig. 8. Dependency Specifics

In the second example, we look at a dependency encountered when designing a model at the enterprise level. A complex dependency, or symbiosis, exists between certain entities within the enterprise, such as the contracts department, the proposal department, and the engineering department. Let us identify one particular dependency that exists at the point of decision on whether or not to bid upon a contract. The contracts and proposal departments depend upon the engineering department for technical knowledge of the scope of the effort, estimates of the cost of performing the contract, estimates of the cost of bidding upon the contract, and possibly estimates of the chances of a contract win. The engineering department and the contracts department depend upon the proposal department for resources and expertise that will allow the preparation of and delivery of a proposal. The engineering and proposal departments depend upon the contracts department for legal knowledge and possibly for a "go/no-go" decision.

This complex dependency may be broken into a set of simple dependencies that explicitly specify each one-way dependency identified. The following list represents some of the dependencies that might be identified:

d1(ContractsDept, ProposalDept)
d2(ContractsDept, EngineeringDept)
d3(ProposalDept, ContractsDept)
d4(ProposalDept, EngineeringDept)
d5(EngineeringDept, ContractsDept)
d6(EngineeringDept,ProposalDept)

We can then determine the attributes associated with each dependency. For example, we could determine that because the enterprise has a documented and often-used policy for proposal preparation, the sensitivity of both d1 and d6 is "robust." However, we also could assign a sensitivity of "fragile" for d4 and d2 if there has been some recent history wherein the engineering department has been less than helpful during proposal efforts. In that case, we could also assign an impact rating of "information unreliable" and an importance rating of medium if the enterprise is of the opinion that the contract bid can take place without detailed engineering input, but would be more confident of their efforts if that detail were available. The stability of d4 could be assigned to "3rd and 4th weeks" if there are two particular weeks when input from the engineering department to the proposal department will make a difference in the proposal department's output. Given the description above, the need associated with d4 could be formulated as a list of capabilities or of outputs from the engineering department required by the proposal department: estimate of technical scope, contract cost estimate, proposal preparation cost estimate, and estimate of win probability. It can also be seen that the modeler might choose to break d4 down further into a set of simple dependencies where the need associated with each is a single product or output.

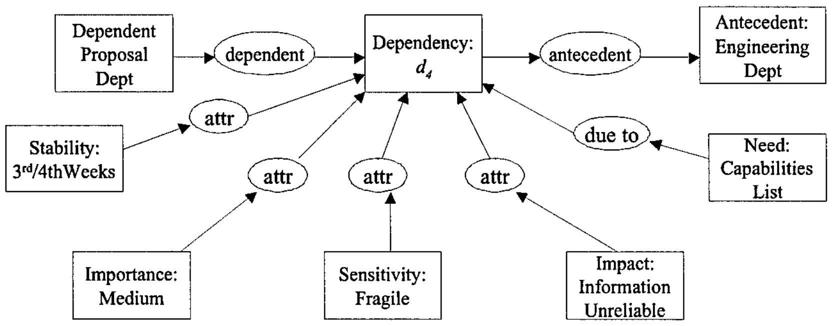


Fig. 9. Dependency Example, Enterprise Level

6.0 Significance

Without a type based approach to dependency analysis, it becomes extremely difficult to distinguish between widely differing dependencies. For example, systems that easily represent functional dependencies have difficulty dealing with causality. The statement that "a dependency exists between" two entities carries almost no information. Is the dependency a causal relationship? Is it a mutually exclusive relationship? Both relationships are dependencies and need to be analyzed in much different ways. A type based approach to dependency analysis will allow the modeling of dependencies at the traditional level where not much is known besides direction of the dependency. But it will also allow much more in-depth analysis of complex relationships in multiple domains.

7.0 Conclusion

The approach given in this paper shows that dependencies can be grouped based upon the identification of attributes applicable to all dependencies. From that set of attributes, a dependency type hierarchy can be produced that will cover all dependencies found in the present literature. Our initial research indicates that this approach provides a more general and unified approach to dependency analysis. We have also shown that Conceptual Graphs provide a powerful approach to represent, characterize, and analyze dependencies between the entities in a model. Using Conceptual Graphs, we can more easily model entities at different levels of model fidelity and when only partial information is available. We are planning continued research which will extend these results to a broader set of practical applications.

8.0 Continuing work

Continuing research is needed to fully investigate and populate the dependency hierarchy in order that all relevant dependency types can be investigated in the approach. We expect that new dependency types will be easily incorporated into our approach simply because it is a type based approach. Also, as we discover more attributes which are pertinent to the general case of dependency, they can be easily added into the existing structure. A more complicated issue which needs to be addressed is the representation and analysis of the most general form of a dependency. A method is needed for decomposing such dependencies in order to simplify analysis. In addition, research is needed in the formal analysis of the method, and in analysis of the complexity of that method. We also need to investigate a more formal definition of our assumptions for entity, change, and potential for change, upon which our definitions are based.

Acknowledgment

This work has been funded in part by the Air Force Research Laboratory, Wright Research Site, for the Defense Advanced Research Agency (DARPA) Autonomic Information Assurance Program.

References

[1] G. Booch, I. Jacobson, J. Rumbaugh, and J. Rumbaugh, The Unified Modeling Language User Guide: Addison-Wesley, 1998.

[2] L. C. Briand, J. Wust, and H. Lounis, "Using Coupling Measurement for Impact Analysis in Object Oriented Systems," presented at IEEE International Conference on Software Maintenance (ICSM98), Bethesda, MD, 1998.

[3] C. Chelba, D. Engle, F. Jelinek, V. Jimenez, S. Khudanpur, L. Mangu, H. Printz, E. Ristad, R. Rosenfeld, A. Stolcke, and D. Wu, "Dependency language modeling, 1996 Large Vocabulary Continuous Speech Recognition Summer Research Workshop," Center for Language and Speech Processing, Johns Hopkins University Technical Reports, Research Note 24, April 15, 1997.

[4] V. M. Crestana-Jensen and A. J. Lee, "Consistent Schema Version Removal: An Optimization Technique for Object Oriented Views," IEEE Transactions on Knowledge and Data Engineering, vol. 12, pp. 261-280, 2000.

[5] R. S. Hall, D. Heimbigner, and A. L. Wolf, "Software deployment languages and schema," Dept. of Computer Science, University of Colorado CU-SERL-203-97, December 18, 1997.

[6] W. J. Hansen, "Deployment Descriptions in a World of COTS and Open Source," Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 1999.

[7] P. Hayes, "Aristotelian and Platonic Views of Knowledge Representations," presented at Second International Conference on Conceptual Structures (ICCS94), College Park, MD, 1994.

[8] A. Keller, U. Blumenthal, and G. Kar, "Classification and Computation of Dependencies for Distributed Management," Proceedings of the Fifth International Conference on Computers and Communications (ISCC 2000), 2000.

[9] A. A. Kountouris and C. Wolinski, "High Level Pre-Synthesis Optimization Steps Using Hierarchical Conditional Dependency Graphs," presented at 25th Euromicro Conference (EUROMICRO '99), Milan, Italy, 1999.

[10] R. Levinson and A. R. Goodwin, "Explorations in Scientific Thinking: a Systems Theoretic Approach: Chapter 2," in Scientific Thinking: a Systems Theoretic Approach. Santa Cruz, CA, 2000, p. 65.

[11] D. Lukose and G. W. Mineau, "A Comparative Study of Dynamic Conceptual Graphs," Brightware Inc/Department of Computer Science, Université Laval,, New York, NY/Quebec City 1998.

[12] G. W. Mineau, "Views, Mappings, and Functions: Exxential Definitions to the Conceptual Graph Theory," presented at Second International Conference on Conceptual Structures (ICCS94), College Park, MD, 1994.

[13] M. Papazoglou, A. Delis, A. Bouguettaya, and M. Haghjoo, "Class Library Support for Workflow Environments and Applications," IEEE TRANSACTIONS ON COMPUTERS, vol. 46, pp. 673-686, 1997.

[14] F. Prost, "A Static Calculus of Dependencies for the l-Cube," presented at 15 th Annual IEEE Symposium on Logic in Computer Science (LICS'00), Santa Barbara, CA, 2000.

[15] J. Rumbaugh, I. Jacobson, and G. Booch, The Unified Modeling Language Reference Manual: Addison-Wesley, 1998.

[16] D. D. K. Sleator and D. Temperley, "Parsing English with a Link Grammar," School of Computer Science Carnegie Mellon University, Pitsburg, PA CMU-CS-91-196, October 1991.

[17] V. S. Subrahamanian, P. Bonatti, J. Dix, T. Eiter, and F. Ozcan, Heterogeneous Agent Systems, 1st ed. Cambridge, Mass: MIT Press, 2000.

[18] B. Thalheim, Entity-Relationship Modeling: Foundations of Database Technology. New York: Springer-Verlag, 1998.

[19] E. S. K. Yu, J. Mylopoulos, and Y. Lespérance, "AI Models for Business Process Reengineering," IEEE Expert, vol. 11, pp. 16-23, 1996.